DIPLOMARBEIT

The DLV$^{\mathcal{K}}$ System for Planning with Incomplete Knowledge

ausgeführt am

Institut für Informationssysteme E 184/3
Abteilung für Wissensbasierte Systeme

der Technischen Universität Wien

unter Anleitung von

O.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Eiter

und

Univ.Ass. Dipl.-Ing. Wolfgang Faber

als verantwortlich mitwirkendem Universitätsassistenten

durch

Axel Florian Polleres,
Klausgasse 35, 1160 Wien

1. Februar 2001

Dedicated to my parents.

Abstract

This thesis presents the Planning System DLV$^{\mathcal{K}}$, which supports the novel Planning Language $\mathcal{K}$. The language allows to represent AI planning problems in a declarative way and is capable of representing incomplete knowledge as well as nondeterministic effects of actions. After explaining some basics, the syntax and semantics of this language will be formally described and some results on the computational complexity of our language will be given, proving that $\mathcal{K}$ is capable of expressing hard planning problems, possibly involving incomplete knowledge or uncertainty, such as secure (conformant) planning.

A translation from various planning tasks specified in $\mathcal{K}$ to a logic programming framework will be shown subsequently. We have implemented a prototype of a planning system, DLV$^{\mathcal{K}}$, on top of the disjunctive logic programming system DLV, to show the practical use of our translation. This prototype will be presented in detail. Finally, examples and experimental results will be given, together with an outlook to further research.

Contents

Acknowledgements

First of all I want to thank my advisors Thomas Eiter, Wolfgang Faber, Nicola Leone and Gerald Pfeifer (in alphabetical order). I was in the lucky situation that in fact four persons supervised my work, though now I can only name two "official" advisors for this thesis. The whole team supported me with all the work for this thesis, and all my (many) open questions were answered within hours, or sometimes minutes, even on weekends, on holidays or late at night. I really benefitted from the interesting work and discussions in this group.

Professor Nicola Leone, who supervised my thesis from the start, kept still in close contact with the project and progress of my work, since he left to Italy last fall, where he is a full professor of Computer Science now in the Department of Mathematics at the University of Calabria. Hopefully this collaboration will continue also after receiving my degree, as I really appreciated the enthusiastic way he lead the project so far.

Wolfgang Faber and Gerald Pfeifer always showed lots of patience with all my questions on implementation. I benefitted a lot from their experience and hints. Special thanks to Wolfgang for proofreading the thesis and also always having an open ear for any theoretical problem, and of course also to Gerald for his patience with my sloppy programming style `;-)`.

I have to thank especially Professor Thomas Eiter, who with his tremendous theoretical knowledge always had a solution for any semantic or syntactic problem developing throughout our work and finally pushed me to finalize the work on my thesis at last, without maybe I still would not have finished it (well, it was about time, I guess `:-)`).

Additionally I want to thank Esra Erdem and Norman McCain from the Texas Action Group in Austin for their helpful explanations on the *Causal Calculator*.

Finally, thanks to all my friends who tolerated my moodiness in the last few weeks, when finalizing this work, and to my parents for supporting me over all the years.

This thesis has been supported by a grant of the FWF (Austrian Science Funds) under the project P11580-MAT.

Chapter 1

Introduction

Planning is an important field in Artificial Intelligence research. Planning enables an agent to synthesize a sequence of actions to achieve a certain goal. Over the last four decades, several approaches to represent and solve planning problems have been developed. For solving problems, translating planning problems into classical logic (SAT-Problems) or logic programs has been proposed ([33],[27],[54],[37],[16]). Among recent approaches high level action languages have been developed, such as the $\mathcal{C}$ action language ([30],[31],[36]), to represent problems in a declarative way. Most of these languages and current translations are based on extensions of classical logic and describe transitions among *possible states of the world* (where each fluent has to be either true or false). However, agents naturally do not have a complete view of the world, and have to reason with *incomplete knowledge*. Nonmonotonic reasoning techniques, like default logic [50] and logic programming under the Answer Set Semantics [26] are well suited to deal with such incomplete information.

The goal of this work was to develop a declarative planning language by extending existing languages. The new language, $\mathcal{K}$, should provide expressive power similar to these existing languages and allow the straightforward translation to an existing formalism capable of nonmonotonic reasoning.

1.1 Terminology

A planning problem in terms of AI consists of

Fluents: Fluents represent time and action dependent predicates, for example `alive`.

Actions: Actions can be performed under certain *preconditions* having *effects* on fluents, for example an action `shoot` can have the effect that the fluent `alive` becomes false under the precondition that the fluent `loaded` is true. Effects of actions can be *deterministic* (`shoot` always makes the fluent `alive` false), or *nondeterministic*.

(Knowledge) States: States are informally a momentary view of the world, i.e. what is known about fluents at a certain point in time. States might be specified *completely* (each fluent is either to be true or false), or *incompletely* (the truth value of some fluents is unknown).

Initial State: The initial state comprises the knowledge of the agent about fluents before starting to act.

Goal: An agent performs actions to achieve a goal, which is a state in which certain fluents are required to be true (resp. false).

Plan: A plan is a sequence of actions that the agent performs to achieve the goal.

We distinguish between *optimistic* and *secure* (conformant) planning: An optimistic plan is a sequence of actions which might achieve the goal, but not necessarily. For instance if the agent should kill somebody (achieve goal `-alive`), performing action `shoot` might be an optimistic plan. It is not secure however, if we do not know whether `loaded` holds initially. A secure plan for this informal example would be to first perform an action `load` followed by `shoot`.

Furthermore, we distinguish between *sequential* planning, where at each point in time only one action might be performed and *concurrent* planning, where parallel actions are allowed.

1.2 Overview

This thesis gives a detailed view on the novel language $\mathcal{K}$, extending the work presented in [15]. Furthermore, the planning system $\texttt{DLV}^{\mathcal{K}}$ is presented, which allows solving hard planning problems, including secure (conformant) planning under incomplete initial knowledge and nondeterministic action effects under certain restrictions. The system allows both sequential and concurrent planning.

Compared with similar action languages, $\mathcal{K}$ is closer in spirit to Answer Set Semantics [26] than to classical logic, allowing to formulate incomplete knowledge and nondeterministic action effects using default negation. The nature of the language allows a straightforward translation of planning domains given in $\mathcal{K}$ to answer set programs. Based on this translation, we have built a planning system on top of the disjunctive logic programming system `DLV`.

After giving a short review of logic programming, Answer Set Semantics and the action language $\mathcal{C}$ in (Chapter 2), a detailed presentation of the language $\mathcal{K}$ follows in Chapter 3. There, syntax, semantics and complexity results for the planning language $\mathcal{K}$ are presented. The subsequent Chapter 4 explains $\mathcal{K}$ programs are translated into logic programs.

After this basic translation a method for checking plans is presented, which allows us to generate secure plans, which achieve the goal even under incomplete

initial knowledge or nondeterministic action effects. Extensions of the translation include sequential planning and an optimization of the translated program, which aims at reducing the grounded logic program.

Chapter 5 discusses the planning system $\texttt{DLV}^{\mathcal{K}}$ and its features. knowledge representation in the language of $\mathcal{K}$ using our prototype is discussed in Chapter 6 where it is shown how to solve some well-known planning problems from the literature using our system. After some experimental results in Chapter 7, the work is concluded giving an outlook to further investigations and research.

Chapter 2

Preliminaries

2.1 Review of Answer Set Programming

In this section, we give a short review of Answer Set Programming (function-free Disjunctive Logic Programming with classical negation and constraints) and point out how Answer Set Programming can be used for temporal reasoning.

Due to [38] a general logic program is a set of rules of the form

$$A_0 \leftarrow A_1, \ldots, A_m, \texttt{not } A_{m+1}, \ldots, \texttt{not } A_n.$$

where $n \geq m \geq 0$, and each A_i is a (function-free) atom, A_0 is called *head* of the rule, the part right of the arrow is called *body*.

Answer Set Programming, first introduced in [26] provides additional features like disjunction in the head, constraints and classical negation "$\neg$" (sometimes referred as "extended" Logic Programming) in addition to negation-as-failure (`not`).

This review is given detailed enough to provide the basis for the further development of a new planning language in the subsequent chapters, which should incorporate similar features.

2.1.1 Syntax of Disjunctive Logic Programs

In the following we will give a short review of Disjunctive Logic Programming with classical negation and constraints.

A *term* in terms of logic programming is a constant or a variable symbol.

An expression of the form $A(t_1, \ldots, t_f)$ is called (function-free) *atom*, where A is a predicate symbols and a possibly empty list of terms $t_1, \ldots, t_f$, also referred as "parameters" sometimes. f is called arity of P.

Notation: *for constant symbols we will use strings beginning with lowercase letters or numbers, for variables strings beginning with an uppercase letter. As predicate symbols we allow any string beginning with a letter.*

A *literal* is an atom $A(t_1, \ldots, t_f)$ or a strongly negated atom $\neg A(t_1, \ldots, t_f)$., where $\neg$ represents classical negation.

A literal/atom is called *ground* if it does not contain any variable symbols.

A *disjunctive logic program* Pi is a set of rules of the form

$$L_1 \vee \ldots \vee L_k \leftarrow L_{k+1}, \ldots, L_m, \texttt{not } L_{m+1}, \ldots, \texttt{not } L_n. \tag{2.1}$$

where $n \geq m \geq k \geq 0$ and L_i $(i = 1, \ldots, m)$ is a literal. If the head of a rule is empty, we denote this rule as "constraint".

A rule is called called *ground* if it contains only ground literals. Consequently a program containing only ground rules is called *ground* as well.

The *Herbrand universe* $\mathcal{HU}_\Pi$ of a program Π is the set of all constant symbols occurring in Π.

A *ground instance* of a rule r finally is a ground rule r' where each variable occurring in r is substituted by an element of $\mathcal{HU}_\Pi$.

2.1.2 Semantics of Disjunctive Logic Programs - Answer Sets

The semantics of disjunctive logic programs treats a rule with variables as a shorthand for the set of its possible ground instances. [1] So it is sufficient to define the semantics only in terms of ground (variable-free) programs.

First we consider only programs without `not` ($m = n$ in every rule 2.1 of the program):

Definition 1. *Let* Π *be a disjunctive logic program without variables and without* `not` *. Let Lit be the set of all ground literals in the language of* Π*. A subset* S *of Lit is called* **answer set** *if*

1. *for any rule* $L_1 \vee \ldots \vee L_k \leftarrow L_{k+1}, \ldots, L_m.$ *from* Π*, if* $L_{k+1}, \ldots, L_m \in S$*, then, for some* $i = 1, \ldots, k$*,* $L_i \in S$ [2]
2. *if* S *contains complementary literals, then* $S = Lit$*.*
3. *there is no smaller subset* $S' \subset S$ *satisfying 1. and 2.*

[1] Note that in many practical answer set systems, such as `DLV`, usually only safe rules are allowed. A rule is safe, if each variable occurring in a head literal or in a body literal preceded by `not` also occurs in at least one body literal which is not preceded by `not` [56]

[2] Whenever the head is empty, this means that for any answer set the following condition must hold: $\{L_{k+1}, \ldots, L_m\} \not\subseteq S$. This way integrity constraints can be modeled.

For example the program

$$a \leftarrow .$$
$$b \vee \neg c \leftarrow a.$$

has two answer sets $\{a, b\}$ and $\{a, \neg c\}$.

Now let Π be an arbitrary disjunctive logic program without variables:

Definition 2. *Let Π be a disjunctive logic program. For any set $S \subseteq Lit$ let Π^S be the program obtained from Π by deleting*

1. *each rule that contains* `not` L *in the body with $L \in S$, and*
2. *all formulae of the form* `not` L *from the bodies of the remaining rules*

As Π^S does not contain default negation `not` *, its answer sets are already defined. If some answer set of Π^S coincides with S then we say that S is an* **answer set** *of Π.*

For example a program consisting of the following rules

$$a \leftarrow \texttt{not}\ b.$$
$$b \vee \neg c \leftarrow a.$$

has the answer set $\{a, \neg c\}$.

2.1.3 Using Answer Set Programming for Temporal Reasoning and Planning

The general idea behind using Answer Set Programming for temporal reasoning is encoding certain facts about the world which can change over time (the *fluents*) and can be influenced by *actions* with predicates extended by an extra integer parameter called time-step. This is basically the same approach as used for the situation calculus [42] to model actions in first-order logic.

For instance, $alive(axel, 0)$ means that *axel* is alive at time 0 (time-step 0 will be referred as *initial state*). The action $shoot(axel, 0)$ would mean that *axel* is shot at time 0. The following example shows how to encode effects of actions:

$alive(axel, 0) \leftarrow .$
$alive(X, T1) \vee \neg alive(X, T1) \leftarrow shoot(X, T), alive(X, T), T1 = T + 1.$
$alive(X, T1) \leftarrow not \neg alive(X, T1), alive(X, T), T1 = T + 1..$

This can be read as follows:

- *axel* is initially *alive*.

- To *shoot* somebody who is *alive* at a certain point in time possibly causes the person to be not *alive* at the next point in time.[3]
- The third rule serves to handle the frame problem [43] in a very elegant way using the expressive power of default negation: Unless there is evidence that a person is not alive, and was alive before, we assume that the person will remain alive.

Although the implementation of action domains sketched here gives an idea of how action/planning domains can be expressed in Answer Set Programming, encoding is not really self-explanatory or easy to read. This is why we will introduce a new planning language called $\mathcal{K}$ in the following, which allows describing planning domains in a more natural way, and subsequently show how this language can be translated to disjunctive logic programs similar to the example above.

***Remark(Notation)**: In the subsequent chapters, whenever using logic programming rules, the prolog/datalog rule operator ":-" will be used instead of "$\leftarrow$" for reasons of consistence with the implementation. By analogy we will use "-" instead of "$\neg$" and "v" instead of "$\vee$" for classical negation and disjunction respectively in examples.*

2.2 Review of $\mathcal{C}$ Action Language

As mentioned before in the introduction there have already been proposals to translate high level action languages to logic programs before, the most recent for action language $\mathcal{C}$, by Lifschitz and Giunchiglia ([30], [28]. A translation for $\mathcal{C}$ to Answer Set Programming is proposed in [36] and [35].

$\mathcal{C}$ more or less served as a starting point for our research, and a major part of our language $\mathcal{K}$, which will be presented subsequently, is based on the work of Lifschitz, Giunchiglia, McCain and Turner. That is why first of all a short review of this action language is given.

The language is based on a set σ of propositional symbols partitioned into σ^{fl} (fluent names) and σ^{act} (action names). Based on these symbols there are two kinds of propositions in $\mathcal{C}$: *static laws* of the form

caused F **if** G

and *dynamic laws* of the form

caused F **if** G **after** U

[3] Disjunction in the head can be used to model nondeterministic effects of actions

where F,G are (propositional) formulae over σ^{fl} and U is a formula over σ. A set of such propositions is called *Action Description*. If the *head* (formula F) of all propositions is either a literal or the symbol $\bot$ and G, U are conjunctions, then the Action Description is called *definite*.

For a definite Action description D a translation $lp_T(D)$ to a logic program is introduced in [35]. T is a positive integer. The language of $lp_T(D)$ has two kinds of atoms: *fluent atoms* - fluent names of D followed by (t) where $t = 0, \ldots, T$ and *action atoms* - action names of D followed by (t) where $t = 0, \ldots, T-1$.

Program $lp_T(D)$ consists of the following rules:

(i) for every static law

caused F **if** $L_1 \wedge \ldots \wedge L_m$

in D we add the rules

$$F(t) \leftarrow \texttt{not } \overline{L_1(t)}, \ldots, \texttt{not } \overline{L_m(t)}.$$

for all $t = 0, \ldots, T$ ($\overline{(L)}$ stands for the complementary literal to L, if $F = \bot$, then $F(t)$ is empty, i.e. the rule has an empty head),

(ii) for every dynamic law

caused F **if** $L_1 \wedge \ldots \wedge L_m$ **after** $L_{m+1} \wedge \ldots \wedge L_n$

in D we add the rules

$$F(t+1) \leftarrow \texttt{not } \overline{L_1(t+1)}, \ldots, \texttt{not } \overline{L_m(t+1)}, L_{m+1}(t), \ldots, L_n(t).$$

for all $t = 0, \ldots, T-1$

(iii) the rules

$\neg B(t) \leftarrow \texttt{not } B(t).$
$B \leftarrow \neg B(t).$

where B is a fluent atom with time stamp 0 and or an action atom.

The last rules represent that occurrence of actions at every point in time, and a complete initial state are "guessed" by the program. Informally, what is shown in [35] is, that the answer sets of this translated program correspond to what we called optimistic plans in the introduction.

In the following we will give a formal description of a language, which extends $\mathcal{C}$ and is more close to Answer Set Programming, allowing a straightforward translation, without the necessity to complete the initial state and incorporating negation as failure in the language itself, which will prove to be very flexible for the representation of actions and planning.

Chapter 3

Planning Language $\mathcal{K}$

3.1 Syntax of $\mathcal{K}$

For our further considerations we use the following disjoint sets:

σ^{act} is a set of action names.

σ^{fl} is a set of fluent names.

σ^{typ} is a set of type names (object types).

These names are effectively predicate symbols with associated arity (≥ 0).

Furthermore, let σ^{con} and σ^{var} be the disjoint sets of constant and variable symbols, respectively.[1]

$\sigma^{fl} \cup \sigma^{act}$ represent the dynamic knowledge whereas σ^{typ}represents the *static background knowledge* (independent from time and actions).

Each $\mathcal{K}$ *program* consists of three parts which will be defined in detail below:

- a disjunction free stratified logic program Π^{typ} (over predicate names in σ^{typ}) called *static background knowledge.*
- an *action description* as defined below.
- a *query* representing a certain goal, which the planning system/agent should achieve.

3.1.1 Actions, Fluents and Types

An action $A \in \sigma^{act}$ (with arity m) can refer to a number of objects, which is written $A(o_1, \ldots, o_n)$ (like an atom in classical logic) and called an *action atom*, in analogy to an action we say A is"executed on the objects " $o_1, \ldots, o_n$,

[1] Following logic programming conventions described in Section 2.1, constant and variable symbols are denoted as strings starting with a lower or uppercase character, respectively.

where $o_1, \ldots, o_n$ may be either variables, or constant symbols from the static background knowledge ($o_1, \ldots, o_n \in \sigma^{con} \cup \sigma^{var}$), for example

```
move(block1,table)
```

to describe an action, where the block `block1` is moved to the table. A fluent $F \in \sigma^{fl}$ (with arity m) can refer to a number of objects, which is written as $F(o_1, \ldots, o_m)$, and called a *fluent atom*, where again $o_1, \ldots, o_m$ are elements of $\sigma^{con} \cup \sigma^{var}$. Fluents describe certain relations among objects which are influenced by actions and time. for example

```
on(block1,block2).
```

describing that the block `block1` is placed on top of `block1`.

Finally, static relations (invariable over the time) among the possible objects are defined through the *type atoms* which represent certain facts that are independent of actions, for example

```
location(table), block(block1), ...
```

Those type predicates representing static knowledge about the world are called the static background knowledge. This knowledge is defined in a disjunction-free, stratified, logic program (For a definition of stratification see [1],[12]).

3.1.2 Literals

An action (resp. fluent, type) *literal* is an action (resp. fluent, type) atom, which is possibly preceded by the true negation symbol "¬". Literals preceded by "¬" are called negative Literals, otherwise positive. A literal (or any other syntactic object) is *ground* if it does not contain variables.

For any literal l, let $\neg.l$ denote its complement, i.e. $\neg l$ if l is an atom and a if $l = \neg a$. Similarly, for a set L of literals, $\neg.L = \{\neg.l \mid l \in L\}$. A set L of literals is *consistent*, if $L \cap \neg.L = \emptyset$. Furthermore, L^+ (resp., L^-) denotes the set of positive (resp., negative) literals in L.

The set of all action (resp. fluent, type) literals is denoted as $\mathcal{L}_{act}$ (resp. $\mathcal{L}_{fl}$, $\mathcal{L}_{typ}$). Furthermore, let then $\mathcal{L}_{fl,typ} = \mathcal{L}_{fl} \cup \mathcal{L}_{typ}$; $\mathcal{L}_{dyn} = \mathcal{L}_{fl} \cup \mathcal{L}^+_{act}$ (*dyn* stands for *dynamic literals*); and $\mathcal{L} = \mathcal{L}_{fl,typ} \cup \mathcal{L}^+_{act}$.[2].

3.1.3 Action/Fluent Declarations

All actions and fluents have to be declared using a statement of the following form:

Definition 3. *A statement of the form:*

$$p(X_1, \ldots, X_n) \texttt{ requires } t_1, \ldots, t_m \tag{3.1}$$

[2]Note that in this definition only positive action literals are allowed. This is because in general there is no intuitive meaning of something like "the opposite of an action occurs"

is called an **action (resp. fluent) declaration** *where* $p \in \mathcal{L}^+_{act}$ *(resp.* $p \in \mathcal{L}^+_{fl}$*),* $X_1, \ldots, X_n \in \sigma^{var}$, $t_1, \ldots, t_m \in \mathcal{L}_{typ}$, *n is the arity of p, and all* X_i *occur also in* $t_1, \ldots, t_m$ *and* $n \geq 0$. *If* $n = 0$, *the* `requires` *part may be skipped.*

That means all variables have to be "bound" to a type in the declaration, for example :

```
move(B,L) requires block(B), location(L)
```

or

```
on(B,L) requires block(B), location(L)
```

3.1.4 Causation Rules

Causation rules are used to define static and dynamic dependencies.

Definition 4. ***Causation rules*** *(**rules**, for short) are of the form*

$$\begin{aligned} &\texttt{caused } f \texttt{ if } b_1, \ldots, b_k, \texttt{not } b_{k+1}, \ldots, \texttt{not } b_l \\ &\qquad \texttt{after } a_1, \ldots, a_m, \texttt{not } a_{m+1}, \ldots, \texttt{not } a_n \end{aligned} \tag{3.2}$$

where $f \in \mathcal{L} \cup \{\texttt{false}\}$, $b_1, \ldots, b_l \in \mathcal{L}_{fl,typ}$, $a_1, \ldots, a_n \in \mathcal{L}$, $l \geq k \geq 0$, *and* $n \geq m \geq 0$. *Rules where* $n = 0$ *are referred to as* ***static rules****, all other rules as* ***dynamic rules*** *as these rules describe changes between states as explained below. When* $l = 0$, *the* `if` *part can be omitted; likewise, if* $n = 0$, *the* `after` *part can be skipped. If both* $l = n = 0$, *also* `caused` *is optional.*

Given a causation rule r, let $h(r) = \{f\}$, $post^+(r) = \{b_1, \ldots, b_k\}$, $post^-(r) = \{b_{k+1}, \ldots, b_l\}$, $pre^+(r) = \{a_1, \ldots, a_m\}$, $pre^-(r) = \{a_{m+1}, \ldots, a_n\}$, and $lit(r) = \{f, b_1, \ldots, b_l, a_1, \ldots, a_n\}$.

Static rules are used to model static state constraints in an action domain.*Dynamic rules* (rules with a non-empty after part) on the contrary can be used to model constraints for state transitions and effects of actions.

3.1.5 Initial State Constraints

While the scope of general static rules is over all knowledge states, it is often useful to specify rules only for the initial states. Static rules can optionally be under initial scope in our language, using the keyword `initially`:

Definition 5. ***Initial state constraints*** *are static rules of the form (3.2) preceded by the keyword* `initially`.

That means the rule only constrain the initial state of knowledge whereas generally rules apply to all states, for example:

```
initially caused f if G
```

[3]. For an initial state constraint ic. $h(ic)$, $post^+(ic)$, $post^-(ic)$, $pre^+(ic)$, and $pre^-(ic)$ are defined as for its rule part.

3.1.6 Constraining Executability of Actions

With the framework defined so far we can forbid actions whenever certain preconditions are NOT fulfilled, using a rule with $h(r) = \texttt{false}$:

$$\texttt{caused false after } a_1, \ldots, a_m, \texttt{not } a_{m+1}, \ldots, \texttt{not } a_n$$

where $a_1 \in \mathcal{L}^+_{act}$ is an action, to express nonexecutability of action a_1 under certain conditions. However, in some cases it would be preferable to be able to express, when an action **is** executable, the simplest example is STRIPS-like [25] planning, where we have a list of preconditions for every rule.

$\mathcal{K}$ allows STRIPS-style conditional execution of actions introducing special rules modeling executability conditions. A difference is that $\mathcal{K}$ allows several alternative executability conditions for an action which are even beyond the repertoire of standard STRIPS. (Though this can be achieved by state of the art planners which accept PDDL[4] planning language as input, an extension of STRIPS where logical formulae are allowed in the precondition. Anyway $\mathcal{K}$offers even more flexibility than these planners.

Definition 6. *An **executability condition** is an expression of the form*

$$\texttt{executable } a \texttt{ if } b_1, \ldots, b_m, \texttt{not } b_{m+1}, \ldots, \texttt{not } b_n \tag{3.3}$$

where $a \in \mathcal{L}^+_{act}$ and $b_1, \ldots, b_n \in \mathcal{L}$, and $n \geq m \geq 0$. If $n = 0$, the $\texttt{if}$ part is usually skipped, expressing unconditional executability.

Given an executability condition e, let $h(e) = \{a\}$, $post^+(e) = post^-(e) = \emptyset$, $pre^+(e) = \{b_1, \ldots, b_m\}$, $pre^-(e) = \{b_{m+1}, \ldots, b_n\}$, and $lit(e) = \{a, b_1, \ldots, b_n\}$.

3.1.7 Safety Restriction

In $\mathcal{K}$, all rules (including initial state constraints) and executability conditions have to satisfy the following syntactic restriction, which is similar to the notion of safety in logic programs [56] (a definition of Safe Datalog can be found in [58]):

[3] If the fluent literal f is ground then it has to be a legal instantiation in terms of the declaration, that means the constants have to correspond with the types given in the declaration for that fluent. (This restriction will be refined later in this section with the definition of a *legal fluent instantiation*)

[4] PDDL is the language used for the AIPS planning competition introduced for AIPS'98, see `ftp://ftp.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz` for the language specification.

Definition 7. *An initial state constraint, a causation rule or an executability condition r is called* ***safe*** *iff all variables in a default negated* **type** *literal t ($t \in (post^-(r) \cup pre^-(r)) \cap \mathcal{L}_{typ}$), also occur in at least one literal which is not a default negated type literal.*

Thus, safety is required only for variables appearing in default negated type literals, while it is not required at all for variables appearing in fluent and action literals. The reason is that the range of the latter variables is implicitly restricted by the respective fluent/action declaration. Also the safety of the rule head in logic programs (corresponding with the `caused` part or `executable` part of a rule or executability condition) is implicit by the range restriction of the fluent/action declarations.

So the difference to the usual notion of safety is that we do not require safety for fluents and action literals since they are typed by definition and therefore already range-restricted by the declaration: We can rewrite all rules and executability conditions with a default negated action or fluent literal in terms of the `requires` clause of the declaration to guarantee safety:

Formally each rule or executability condition r is transformed to a new rule/executability condition r' with the same head ($h(r') = h(r)$) where pre^+(r) is extended with the substituted requires part of the fluent/action declaration (see 3.1) of the fluent/action in the head: That means, assuming $h(r) = b(\overline{X})$, let

$$b(\overline{Y}) \texttt{ requires } t_1(\overline{Y}_1), \ldots, t_m(\overline{Y}_m)$$

be the corresponding declaration. Then

$$pre^+(r') = pre^+(r) \cup \theta(t_1(\overline{Y}_1)), \ldots, \theta(t_m(\overline{Y}_m))$$

where θ is a substitution, such that $\theta(\overline{Y}) = \overline{X}$. If $h(r) = \texttt{false}$ then $pre^+(r') = pre^+(r)$.

Additionally for each fluent/action in pre^-(r), i.e. for all default negated actions/fluents, we have to add the substituted requires part of the corresponding action/fluent declarations to pre^+(r'). This yields the following new pre^+(r'): Let $p_i \in pre^-(r) \setminus \mathcal{L}_{typ} = \{p_1, \ldots, p_n\}$ where $p_i = b_i(\overline{X}_i)$ has a declaration of the form:

$$b_i(\overline{Y}_i) \texttt{ requires } t_{1,i}(\overline{Y}_{1,i}), \ldots, t_{m_i,i}(\overline{Y}_{m_i,i})$$

then

$$pre^+(r') = pre^+(r') \cup \bigcup_{i \in \{1,\ldots,n\}} \{\theta(t_{1,i}(\overline{Y}_{1,i})), \ldots, \theta(t_{m_i,i}(\overline{Y}_{m_i,i}))\}$$

where θ is a substitution, such that $\theta(\overline{Y}_i) = \overline{X}_i$ for all $i \in \{1, \ldots, n\}$.
Analogously for each fluent in $post^-$(r) (the default negated fluents), we have to add the requires part to $post^+$(r'): Let $p_i \in post^-(r) \setminus \mathcal{L}_{typ} = \{p_1, \ldots, p_n\}$ where $p_i = b_i(\overline{X}_i)$ has a declaration

$$b_i(\overline{Y}_i) \texttt{ requires } t_{1,i}(\overline{Y}_{1,i}), \ldots, t_{m_i,i}(\overline{Y}_{m_i,i})$$

then

$$post^+(r') = post^+(r) \cup \bigcup_{i \in \{1,\ldots,n\}} \{\theta(t_{1,i}(\overline{Y}_{1,i})), \ldots, \theta(t_{m_i,i}(\overline{Y}_{m_i,i}))\}$$

where θ is a substitution, such that $\theta(\overline{Y}_i) = \overline{X}_i$ for all $i \in \{1, \ldots, n\}$.
Finally $post^-$(r) and pre^-(r) remain unchanged:

$post^-(r') = post^-(r)$
$pre^-(r') = pre^-(r')$

As the rule/executability condition r' is uniquely characterized by h(r'), pre^+(r'), pre^-(r'), $post^+$(r') and $post^-$(r') the rewriting is now completely defined.

So the safety restriction defined above is always understood w.r.t. this rewriting.

Example: Given an action `move` and fluents `on`, `occupied` with declarations

```
move(B,L) requires block(B), location(L).
on(B1,L) requires block(B), location(L).
occupied(B) requires location(B).
```

the executability condition

```
executable move(B1,L1) if not occupied(B1), not occupied(L1),
                          B1 ≠ L1.
```

and the causation rule

```
caused on(B1,L1) after move(B1,L1).
```

are rewritten to:

```
executable move(B1,L1) if not occupied(B1), not occupied(L1),
                          B1 ≠ L1, block(B1), location(L1),
                          location(B1).
caused on(B2,L2) after move(B2,L2), block(B2), location(L2).
```

3.1.8 Action Descriptions, Planning Domains and Planning Problems

Definition 8. *An **action description** is a pair $\langle D, R\rangle$ where D is a finite set of action and fluent declarations and R is a finite set of safe executability conditions, safe causation rules, and safe initial state constraints, s.t. there exists a declaration for each action or fluent name occurring in R.*

Definition 9. *A **planning domain** is a pair $PD = \langle \Pi, AD\rangle$, where Π is a normal stratified datalog program (referred to as **background knowledge**) that is safe (in the standard LP sense), and AD is an action description.*

PD is positive, *if no default negation occurs in AD.*

Definition 10. *A **query** is of the form*

$$g_1, \ldots, g_m, \texttt{not } g_{m+1}, \ldots, \texttt{not } g_n \ ?\ (i) \tag{3.4}$$

where $g_1, \ldots, g_n \in \mathcal{L}_{fl}$ are variable-free, $n \geq m \geq 0$ and $i \geq 0$.

The list of literals $g_1, \ldots, g_m, \texttt{not } g_{m+1}, \ldots, \texttt{not } g_n$ is called the **goal** and `i` is a positive integer which is called **plan length**. The query can be read as: "Is there a plan achieving the goal in `i` steps?"

Definition 11. *A **planning problem** $\langle PD, q \rangle$ is a pair of a planning domain PD and a query q.*

3.2 Semantics of $\mathcal{K}$

First we will define the legal instantiations of a planning problem. This is similar to the grounding of a logic program—the difference is that only correctly typed fluent and action literals are generated.

3.2.1 Instantiation

Let $PD = \langle \Pi, \langle D, R \rangle \rangle$ be a $\mathcal{K}$ planning domain, and let M be the (unique, finite) answer set of the static background knowledge Π.

Let substitutions and their application to syntactic objects be defined as usual (assignments of constants to variables). We first define the notion of legal action (resp. fluent) instantiations:

Definition 12. *For each action (resp. fluent) p given the declaration*

$$p(\overline{X}) \texttt{ requires } t_1, \ldots, t_m$$

*in D, let θ be a substitution. Then $\theta(p(\overline{X}))$ is a is a **legal action** (resp. **fluent**) **instantiation**, if θ is a (ground[5]) substitution defined over $\overline{X}$ such that $\{\theta(t_1), \ldots, \theta(t_m)\} \subseteq M$. By $\mathcal{L}_{PD}$ we denote the set of all legal action and fluent instantiations.*

Legal Instantiations of Rules

Legal instantiations of rules and initial state constraints are defined as above with the following difference: Substitutions of the default negated literals in the `if` or `after` part are not restricted to legal action/fluent instantiations. Still the legal instantiations remain finite and unique, as the instantiations of those default negated literals are uniquely determined by the *safety restriction* (see section 3.1.7.

Definition 13. *Let r be a safe $\mathcal{K}$ causation rule or initial state constraint:*

[5] i.e. a substitution which substitutes all variables with constants

$$\begin{aligned}\texttt{caused } f(\overline{X}) \texttt{ if } & b_1(\overline{X_1}), \ldots, b_{m^{pos}}(\overline{X_{m^{pos}}}),\\ & \texttt{not } b_{m^{pos}+1}(\overline{X_{m^{pos}+1}}), \ldots, \texttt{not } b_m(\overline{X_m})\\ \texttt{after } & b_{m+1}(\overline{X_{m+1}}), \ldots, b_{n^{pos}}(\overline{X_{n^{pos}}}),\\ & \texttt{not } b_{n^{pos}+1}(\overline{X_{n^{pos}+1}}), \ldots, \texttt{not } b_n(\overline{X_n})\end{aligned}$$

Let θ be a substitution. Then the rule r'

$$\begin{aligned}\texttt{caused } \theta(f(\overline{X})) \texttt{ if } & \theta(b_1(\overline{X_1})), \ldots, \theta(b_{m^{pos}}(\overline{X_{m^{pos}}})),\\ & \texttt{not } \theta(b_{m^{pos}+1}(\overline{X_{m^{pos}+1}})), \ldots, \texttt{not } \theta(b_m(\overline{X_m}))\\ \texttt{after } & \theta(b_{m+1}(\overline{X_{m+1}})), \ldots, \theta(b_{n^{pos}}(\overline{X_{n^{pos}}})),\\ & \texttt{not } \theta(b_{n^{pos}+1}(\overline{X_{n^{pos}+1}})), \ldots, \texttt{not } \theta(b_n(\overline{X_n}))\end{aligned}$$

is a **legal rule instantiation** *of the rule r iff*

1. *$\theta(f(\overline{X}))$ is a legal fluent instantiation.*
2. *for each $b_i \in \mathcal{L}_{dyn} : \theta(b_i(\overline{X_i}))$ is a legal action/fluent instantiation, where $i \in \{1, \ldots, m^{pos}\} \cup \{m+1, \ldots, n^{pos}\}$*
3. *for each $b_j \in \mathcal{L}_{typ} : \theta(b_j(\overline{X_j}))$ is in M, where $j \in \{1, \ldots, n\}$*

Legal Instantiation of Executability Conditions

The legal instantiations of executability conditions are defined analogously:

Definition 14. *Let c be a safe $\mathcal{K}$ executability condition:*

$$\begin{aligned}\texttt{executable } a(\overline{X}) \texttt{ if } & b_1(\overline{X_1}), \ldots, b_m(\overline{X_m}),\\ & \texttt{not } b_{m+1}(\overline{X_{m+1}}), \ldots, \texttt{not } b_n(\overline{X_n})\end{aligned}$$

Let θ be a substitution. Then the executability condition c'

$$\begin{aligned}\texttt{executable } \theta(a(\overline{X})) \texttt{ if } & \theta(b_1(\overline{X_1})), \ldots, \theta(b_m(\overline{X_m})),\\ & \texttt{not } \theta(b_{m+1}(\overline{X_{m+1}})), \ldots, \texttt{not } \theta(b_n(\overline{X_n}))\end{aligned}$$

is a **legal instantiation** *of executability condition c iff all the following conditions hold:*

1. *$\theta(a(\overline{X}))$ is a legal action instantiation.*
2. *for each $b_i \in \mathcal{L}_{dyn} : \theta(b_i(\overline{X_i}))$ is a legal action/fluent instantiation, where $i \in \{1, \ldots, m\}$*
3. *for each $b_j \in \mathcal{L}_{typ} : \theta(b_j(\overline{X_j}))$ is in M, where $j \in \{1, \ldots, n\}$*

Finally: The **instantiation of a planning domain** $PD = \langle \Pi, \langle D, R \rangle \rangle$ consists of all possible legal instantiations of R according to the declarations in D, i.e.:

- The set of all legal instantiations of the initial state constraints (static rules under initial scope).
- The set of all legal instantiations of the causation rules.
- The set of all legal instantiations of executability conditions.

Using a more formal compact description we can define the instantiation of a planning domain as follows:

Definition 15. *Given a planning domain* $PD = \langle \Pi, \langle D, R \rangle \rangle$ *where* M *is the unique answer set of* Π

$$PD\downarrow = \bigcup_{r \in R} \bigcup_{\theta \in \Theta_r} \{\theta(r)\},$$

where Θ_r *is the set of all substitutions* θ *defined over all variables in* r*, such that* $lit(\theta(r)) \cap \mathcal{L}_{dyn} \subseteq \mathcal{L}_{PD}$ *and* $(post^+(\theta(r)) \cup pre^+(\theta(r))) \cap \mathcal{L}_{typ} \subseteq M$ *hold. In other words, actions and fluents must agree with their declarations and positive type literals must agree with the background knowledge.*

$PD\downarrow$ has a ground action description, in which all fluent and action literals agree with their declarations and all type literals agree with the background knowledge.

This instantiation makes the planning problem ground in the sense of logic programming and allows us to define the semantics in terms of a ground program without loss of generality:

So in the following, whenever we speak about literals, rules, action descriptions and planning domains respectively, we refer to legal instantiations. The semantics of a general planning problem with variables is determined by its instantiation.

3.2.2 States and State Transitions

In analogy to the definition of stable models and answer sets [26] (see Chapter 2.1), we will first define the semantics for positive (i.e., default negation free) planning problems. Subsequently we define a reduction from general planning problems to positive ones.

In what follows, let PD be a planning domain with instantiation $PD\downarrow = \langle \Pi, \langle D, R \rangle \rangle$, and M be the unique answer set of Π.

States and State Transitions in Positive Planning Domains

Definition 16. *A consistent set of ground fluent literals is called* ***state****. A tuple* $t = \langle s, A, s' \rangle$ *where* s, s' *are states and* A *is a set of action atoms is called a* ***state transition****.*

Given (the legal instantiation of) a positive planning domain, a minimal consistent set s_0 of fluent literals which satisfies the initial state constraints is called a legal initial state:

Definition 17. *A state s_0 is called **legal initial state** for a positive PD iff for each initial state constraint and static rule respectively $c \in R$, $h(c)$ is in s_0 if $post^+(c) \subseteq s_0 \cup M$ holds and s_0 is minimal under this condition.*

In other words for each initial state constraint or causation rule in *PD*

`(initially) caused` f `if` $b_1, \ldots, b_n$

f is in s_0 if

1. for all $b_i \in \mathcal{L}_{fl} : b_i \in s_0, i \in \{1, \ldots, n\}$
2. for all $b_j \in \mathcal{L}_{typ} : M \models b_j \; j \in \{1, \ldots, n\}$.

Given a positive planning domain and a state s, a set of action atoms $A = \{a_1, \ldots, a_k\}$ is called **executable action set** w.r.t. state s iff for each a_i in A there exists an executability condition

`executable` a_i `if` $b_1, \ldots, b_m, a_{i_1}, \ldots, a_{i_n}$

such that

1. $\{b_1, \ldots, b_m\} \subseteq s \cup M$ and
2. $a_{i_j} \in A$ for all $j \in \{1, 2, \ldots, n\}$.[6]

This leads to the following definition:

Definition 18. *For a positive PD and a state s, a set $A \subseteq \mathcal{L}^+_{act}$ is called **executable action set** w.r.t. s iff for each $a \in A$ there exists an executability condition $e \in R$ such that $h(e) = \{a\}$, $pre^+(e) \subseteq s \cup A \cup M$.*

Definition 19. *For a positive PD a state transition $t = \langle s, A, s' \rangle$ is called **legal state transition** if A is an executable action set w.r.t. s and s' is a minimal consistent set that satisfies all causation rules w.r.t. $s \cup A \cup M$, i.e. for every causation rule $r \in R$, if (i) $post^+(r) \subseteq s' \cup M$, (ii) $pre^+(r) \subseteq s \cup A \cup M$ hold, then $h(r) \neq \{\texttt{false}\}$ and $h(r) \in s'$.*

In other words for each causation rule:

`caused` f `if` $b_1, \ldots, b_m$ `after` $b_{m+1}, \ldots, b_n$

f is in s' if

1. for all $b_i \in \mathcal{L}_{fl} : b_i \in s', i \in \{1, \ldots, m\}$ and
2. for all $b_j \in \mathcal{L}_{fl} : b_j \in s, j \in \{m+1, \ldots, n\}$ and
3. for all $b_k \in \mathcal{L}_{typ} : M \models b_k, k \in \{1, \ldots, n\}$ and

[6] This is useful to model compound actions, i.e. actions which may only occur together with special other actions. If we disallow concurrent actions (see Section 3.2.6) clearly the occurrence of other actions in the `if` part of an executability condition does not make sense.

4. for all $b_l \in \mathcal{L}_{act} : b_l \in A, l \in \{m+1, \ldots, n\}$.

As this definition leads to an inconsistent state when the head of a static/dynamic rule is `false` this can be used to formulate integrity constraints, which rule out certain constellations of fluents/actions, for instance:

```
caused false after move(B,L), move(B,L1), -equal(L,L1)
```

expresses that one block can not be moved to two different locations simultanously.

3.2.3 Reduction

Definition 20. *For an arbitrary PD and a state transition $t = \langle s, A, s' \rangle$, the* **reduction** *$PD^t = \langle \Pi, \langle D, R^t \rangle \rangle$ is a planning domain where R^t is obtained from R by deleting those $r \in R$, for which either $post^-(r) \cap (s' \cup M) \neq \emptyset$ or $pre^-(r) \cap (s \cup A \cup M) \neq \emptyset$ holds, and by deleting all* `not` *L ($L \in \mathcal{L}$) from the remaining $r \in R$. Note that PD^t is positive and ground.*

That means the reduction is the positive problem domain obtained from a general problem domain by deleting:

1. All causation rules and initial state constraints which contain `not` f in the `if` part and $f \in s'$ (for $f \in \mathcal{L}_{fl}$).
2. All dynamic causation rules which contain `not` f in the `after` part and $f \in s$ (for $f \in \mathcal{L}_{fl}$).
3. All dynamic causation rules which contain `not` a_i in the `after` part and $a_i \in A$ (for $a_i \in \mathcal{L}_{act}$).
4. All causation rules and initial state constraints resp. which contain `not` b in the `if` part or after part and $b \in M$ (for $b \in \mathcal{L}_{typ}$).
5. all executability conditions which contain `not` f in the `if` part and $f \in s$ (for $f \in \mathcal{L}_{fl}$).
6. all executability conditions which contain `not` a_i in the `if` part and $a_i \in A$ (for $a_i \in \mathcal{L}_{act}$).
7. all expressions of the form `not` L (for $L \in \mathcal{L}$) from the remaining causation rules and executability conditions.

3.2.4 States and State Transitions In General

Definition 21. *For an arbitrary PD, a state s_0 is called* ***legal initial state*** *iff s_0 is a legal initial state for PD^t with $t = \langle \emptyset, \emptyset, s_0 \rangle$.*[7]

[7] Note that in contrast to the positive definition now there can be several legal initial states due to the nonmonotonic effects of using default negation.

*A set of action atoms A is an **executable action set** in PD w.r.t. a state s iff A is executable w.r.t. s in PD^t with $t = \langle s, A, \emptyset \rangle$.*
A transition $t = \langle s, A, s' \rangle$ is a **legal state transition** *in PD iff it is a legal transition w.r.t PD^t.*[8]

3.2.5 Transition Sequences and Plans

Definition 22. *A sequence of legal state transitions $T = \langle \langle s_0, A_0, s_1 \rangle, \ldots, \langle s_{n-1}, A_{n-1}, s_n \rangle \rangle$, $n \geq 0$, is a **legal transition sequence** for PD, if*

1. *s_0 is a legal initial state of PD and*
2. *all $\langle s_{i-1}, A_{i-1}, s_i \rangle$, $1 \leq i \leq n$, are legal state transitions of PD.*

In particular $T = \langle\rangle$ is empty if $n = 0$.

Optimistic Plans

This directly leads to the following simple definition of a plan:

Definition 23. *Given a planning problem $PP = \langle PD, q \rangle$, where q has form (3.4), a sequence of action sets $\langle A_0, \ldots, A_i \rangle$, is an **(optimistic) plan** for PP, if a legal transition sequence $T = \langle \langle s_0, A_0, s_1 \rangle, \ldots, \langle s_{i-1}, A_{i-1}, s_i \rangle \rangle$ in PD exists such that T establishes the goal, i.e.,*

1. *$\{g_1, \ldots, g_m\} \subseteq s_i$ and*
2. *$\{g_{m+1}, \ldots, g_n\} \cap s_i = \emptyset$*

Secure Plans

However, the existence of an optimistic plan does not guarantee that executing the plan, due to incomplete information and possible alternative transitions, will always lead to the goal deterministically. To achieve that, we need a more strict definition that guarantees determinism of plan execution:

An optimistic plan $p = \langle A_0, \ldots, A_{n-1} \rangle$ is called **secure plan** iff for all possible legal initial states s_0 executability of p is guaranteed and leads to the goal:

Definition 24. *An optimistic plan $\langle A_0, \ldots, A_i \rangle$ for PP as previously is a **secure plan**, if for every legal initial state s_0 and legal transition sequence $T = \langle \langle s_0, A_0, s_1 \rangle, \ldots, \langle s_{j-1}, A_{j-1}, s_j \rangle \rangle$ such that $0 \leq j \leq i$, it holds that*

1. *if $j = i$ then T establishes the goal, and*
2. *if $j < i$, then A_j is executable in s_j w.r.t. PD, i.e., some legal transition $\langle s_j, A_j, s_{j+1} \rangle$ exists.*

[8] Also here now several different successor states can be possible due to the nonmonotonicity of default negation where in the positive definition the successor state was uniquely determined by the actions.

3.2.6 Concurrent Actions

Our definition of transitions allows concurrent actions, that means more than one atomic action may occur at once. Sometimes it is desired in planning to prohibit concurrent actions to gain sequential plans. For this purpose we define the set of all legal state transitions *under no concurrency* as follows:

If T denotes the set of all legal state transitions $t = \langle s, A, s' \rangle$ then $T^{noconc} \subseteq T$ denotes the set of all legal state transitions where A has at most one element ($|A| \leq 1$). $\mathcal{K}^{noconc}$ denotes the fragment of $\mathcal{K}$ which allows only atomic actions.

For the nonconcurrent case an optimistic (resp. secure) plan p^{noconc} may consist only of atomic actions, that means $|A_0|, |A_1|, \ldots, |A_{i-1}| \leq 1$, which implies that all state transitions t to carry out the plan have to be in $\mathcal{K}^{noconc}$: Such a plan $\langle A_0, \ldots, A_{i-1} \rangle$ is called **sequential**.

3.2.7 Language Enhancements

In this section we define some shortcuts to express inertia, default knowledge, nonexecutability of actions and for modeling incomplete knowledge.

Default Knowledge

First we define a shortcut for modeling default knowledge, assumptions which are true in the absence of more specific knowledge, using the keyword `default`. This statement may also be used under initial scope (modeling default knowledge in the initial state):

`(initially) default` f.

which is equivalent to the (static) causation rule

`(initially) caused` f `if not` $\neg.f$.

where f is a fluent literal.

Inertia

In planning it is often useful to declare some fluents as inertial, which means that these fluents keep their truth values in a state transition, unless explicitly affected by an action. Inertia of fluents is strongly related to the *frame problem* [43, 51], which has been studied intensively in the AI literature.

To allow for an easy representation of inertia, we have enhanced the language with a shortcut

`inertial` f.

which is equivalent to the (dynamic) causation rule

`caused` f `if not` $\neg.f$ `after` f.

where f is a fluent literal.

Reasoning under Incomplete Knowledge

For reasoning under incomplete knowledge we introduce totalization rules (possibly under initial scope) to model completion of incomplete knowledge of an agent about a fluent:

$$\begin{aligned}(\texttt{initially})\ \texttt{total}\ f\ \texttt{if}\ b_1,\ldots,b_k,\texttt{not}\ b_{k+1},\ldots,\texttt{not}\ b_l\\ \texttt{after}\ a_1,\ldots,a_m,\texttt{not}\ a_{m+1},\ldots,\texttt{not}\ a_n.\end{aligned}$$

which is a shortcut for

$$\begin{aligned}(\texttt{initially})\ \texttt{caused}\ f\ \texttt{if not}\ \neg f,b_1,\ldots,b_k,\texttt{not}\ b_{k+1},\ldots,\texttt{not}\ b_l\\ \texttt{after}\ a_1,\ldots,a_m,\texttt{not}\ a_{m+1},\ldots,\texttt{not}\ a_n.\\ (\texttt{initially})\ \texttt{caused}\ \neg f\ \texttt{if not}\ f,b_1,\ldots,b_k,\texttt{not}\ b_{k+1},\ldots,\texttt{not}\ b_l\\ \texttt{after}\ a_1,\ldots,a_m,\texttt{not}\ a_{m+1},\ldots,\texttt{not}\ a_n.\end{aligned}$$

where f is a positive fluent literal. Such `total` statements if preceded by keyword `initially`, only complete the initial state. In this case the `after` part has to be empty ($m = n = 0$).[9] These rules guarantee total state information w.r.t. a given fluent. This means that for any fluent where no other information exists a truth value is "guessed", which allows us to generate plans in which the value of an undefined fluent is essential.

Furthermore, we can restrict guessing to the case when some preconditions hold in the current state ($b_1,\ldots,b_k,\texttt{not}\ b_{k+1},\ldots,\texttt{not}\ b_l$) and model nondeterministic changes over a transition ($a_1,\ldots,a_m,\texttt{not}\ a_{m+1},\ldots,\texttt{not}\ a_n$).

So the `total` statement forces to guess the truth value of a fluent, if it is unknown in a state. In this sense $\mathcal{K}$ allows conditional totalization of fluents. As we will see in the subsequent chapters and the examples later on, the `if` part in totalization rules allows more compact modeling of some examples, and more efficient translation to logic programming. This is especially helpful in the initial case to "guess" in case of incomplete initial knowledge, but also to model nondeterministic effects of actions.

Nonexecutability of Actions

Finally, it may be convenient to explicitly forbid executing an action under specific circumstances. To this end, we introduce

$$\texttt{nonexecutable}\ a\ \texttt{if}\ B.$$

where a is an action atom, as a shortcut for

$$\texttt{caused false after}\ a,B.$$

In case of conflicts, `nonexecutable A` overrides `executable A`.

[9] Note that this is a remarkable difference to Lifschitz' $\mathcal{C}$ Action Language [30, 36, 35] where the totality of fluents in all states is implicit. In our language a fluent must be declared to be possible incomplete in using this `total` primitive. This approach on the other hand is closer to Answer Set Programming whereas $\mathcal{C}$ Action Language is oriented towards planning as satisfiability, due to Kautz and Selman [33], where total interpretations are implicit and there is no possibility for conditional totality of fluents.

3.3 Complexity of $\mathcal{K}$

In this section, we present some results on the computational complexity of planning in $\mathcal{K}$ for the ground (propositional) case (see [2, 9] and references therein for related results).[10] In particular, we consider here the following three problems:

- Deciding, given a propositional planning problem $\langle PD, q\rangle$, whether some optimistic plan exists.
- Deciding, given an optimistic plan $P = \langle A_1, \ldots, A_n\rangle$ for a propositional planning program $\langle PD, q\rangle$, whether P is secure.
- Deciding, given a propositional planning problem $\langle PD, q\rangle$, whether some secure plan exists.

We say that an arbitrary planning domain PD is *proper* if, given a state s and an action set A, the existence of a legal state transition $\langle s, A, s'\rangle$ is polynomially decidable (i.e., we can check efficiently the existence of a successor state s'). A planning problem $\langle PD, q\rangle$ is *proper* if the underlying planning domain PD is proper.

We call a legal transition $\langle s, A, s_1\rangle$ *determined*, if $s_1 = s_2$ holds for every legal transition $\langle s, A, s_2\rangle$ (i.e., executing A on s leads to a unique new state), and we call a planning domain *deterministic*, if all legal transitions in it are determined.

The existence of a secure plan can be decided by composing an algorithm for constructing optimistic plans and an algorithm for checking security of an optimistic plan (this is actually the algorithm of the current prototype, presented in the subsequent chapters). Our membership proofs for deciding the existence of an optimistic plan actually (nondeterministically) construct such a plan, and thus we can easily derive upper bounds on the complexity of deciding the existence of a secure plan from the complexity of the combined algorithm. (We will repeatedly use slightly simpler algorithms, though, which do not actually generate optimistic plans in the first step.)

We remark here that the formulation of security checking is, strictly speaking, a *promise problem*, since it is *asserted* that P is an optimistic plan, which can not be checked in polynomial time in general (and thus legal inputs can not be recognized easily). However, the complexity results that we derive below would remain the same, even if P were not known to be an optimistic plan.

We will consider the three problems from above under the following two restrictions:

Proper vs general planning domains We pay special attention to proper planning domains, and contrast them to general planning domains.

[10] The results and proofs given here are taken from [17].

In fact, our proofs of the lower bounds for proper planning domains apply to a particular syntactic subclass, which is defined as follows. We call a planning domain $PD = \langle \Pi, AD \rangle$ *plain*, if the background knowledge Π is empty, and AD satisfies the following conditions:

- Executability conditions refer only to fluents.
- No default negation –neither explicit nor implicit through language extensions– is used in the *post*-part of causation rules in the `'always'` section. In particular, inertia rules are disallowed.
- Given that $\alpha_1, \ldots, \alpha_m$ are all ground actions, AD contains the rules

  `caused false after` α_i`,` α_j $\quad 1 \leq i < j \leq m$
  `caused false after not` α_1`, not` α_2`, ..., not` α_m `.`

  They ensure that in each legal state transition $t = \langle s, A, s' \rangle$, $|A| = 1$ must hold. Thus, `noConcurrency` is implicitly enforced, and each optimistic plan (thus also each secure plan) must be sequential.

It is easy to see that under these conditions, given some state s and an action set A, deciding whether some legal transition $\langle s, A, s' \rangle$ exists is polynomial. In other words, *PD* is proper. Moreover, each legal transition is clearly determined, and thus *PD* is also deterministic.

We call a planning problem $PP = \langle PD, q \rangle$ *plain* if *PD* is plain.

Fixed vs arbitrary plan length We analyze the impact of fixing the length i in the query $q = Goal\ ?\ (i)$ of $\langle PD, q \rangle$ to a constant.

Note that, in general, the length of a possible plan for $\langle PD, q \rangle$ can be exponential in the size of the string representing i (which, as usual, is represented in binary), and also exponential in the size of the string representing the whole input $\langle PD, q \rangle$. Thus, storing a complete possible plan in the working memory requires exponential space in general. If i is fixed, however, then the representation size of the plan is linear in the size of $\langle PD, q \rangle$.

Properness even holds if we allow, what will be called "**stratified**" default negation in the *post*-part of causation rules in the `'always'` section: For this we need to adapt the definition of stratified logic programs to our language:

Definition 25. *A planning domain PD is* stratified, *if there is an assignment $str(i)$ of integers $0, 1, \ldots$ to the fluent literals in PD (=$\mathcal{L}_{fl}$) such that for each causation rule r in PD the following holds: Taken $h(r) = f$, if $g \in post^+(r)$ then $str(f) \geq str(g)$ and if $g \in post^-(r)$ then $str(f) > str(g)$.*

The problem of deciding whether some legal transition $\langle s, A, s' \rangle$ exists can, be reduced to a the problem whether a logic program has a stable model, the translation will be shown in the next chapter. For stratified planning domains

this translated logic program is stratified in the sense of logic programming. Stratified logic programs, however, always have a unique stable model, so to decide whether a legal transition exists is trivial and therefore stratified planning domains are proper.

Allowing stratified default negation especially affects inertia rules, where default negation occurs in the *post*-part of causation rules:

`caused` f `if not` $\neg.f$ `after` f.

Plain programs would not even allow any form of inertia, whereas we can allow inertia, as long as the domain remains stratified, without hurting properness.

Determinism though does not hold anymore, as inertia rules can be used to "inherit" nondeterminism to subsequent states, i.e.

total a.

can be emulated for the fixed-length case with the following translation for given plan length n:

`initially total` $guess_1$.
`initially total` $guess_2$.
$\vdots$
`initially total` $guess_n$.
`inertial` $guess_1$.
`inertial` $guess_2$.
$\vdots$
`inertial` $guess_n$.
`initially` t_1.
`caused` t_2 `after` t_1.
$\vdots$
`caused` t_n `after` t_{n-1}.
`initially total` a.
`caused` a `after` $guess_1, t_1$.
`caused` $-a$ `after not` $guess_1, t_1$.
$\vdots$
`caused` a `after` $guess_n, t_n$.
`caused` $-a$ `after not` $guess_n, t_n$.

Obviously this translation is polynomial and based only on inertia-rules, which do not hurt stratification. This translation shows furthermore that properness is not influenced by allowing `total` statements for the fixed-length case.

3.3.1 Results

In the derivation of the lower bounds of our results, the background knowledge Π of the planning domain $PD = \langle \Pi, AD \rangle$ will always be empty.

We start with noting the following auxiliary result on checking initial states and legality of state transitions.

Lemma 1. *Given a state s_0 (resp., a state transition $t = \langle s, A, s' \rangle$) and a propositional planning domain $PD = \langle \Pi, \langle D, R \rangle \rangle$, checking whether s_0 is a legal initial state (resp., t is a legal state transition) is possible in polynomial time.*

Proof. The unique answer set M of the stratified normal logic program Π can be computed in polynomial time (cf. [12]). Given M, the reduction PD^t can easily be computed in polynomial time. Checking whether s_0 is a legal initial state for PD^t amounts to checking whether s_0 is the least fix-point of a set of positive ground rules, which is well-known polynomial. Overall, this means that checking whether s_0 is a legal initial state of PD is polynomial. From M, t, and PD^t, it can be easily checked in polynomial time whether A is executable w.r.t. s and, furthermore, whether s' is the minimal consistent set that satisfies all causation rules w.r.t. $s \cup A$ by computing the least fixpoint of a set of positive rules and verifying constraints on it. Thus, checking whether t is a legal state transition is polynomial in the propositional case. □

Corollary 1. *Given a sequence of state transitions $T = \langle t_1, \ldots, t_n \rangle$, where $t_i = \langle s_{i-1}, A_i, s_i \rangle$ for $i = 1, \ldots, n$, and a propositional planning domain $PD = \langle \Pi, \langle D, R \rangle \rangle$, checking whether T is legal w.r.t. PD is possible in polynomial time.*

An optimistic plan $P = \langle A_1, \ldots, A_n \rangle$ can be generated nondeterministically, by guessing legal transitions $\langle s_{i-1}, A_i, s_i \rangle$ subsequently, starting from some (nondeterministically generated) legal initial state s_0. Since this requires only polynomial workspace and NPSPACE = PSPACE, the problem is in PSPACE. On the other hand, propositional STRIPS, which is PSPACE-complete [9], can be easily reduced to $\mathcal{K}$:

Theorem 1. *Deciding whether for a given ground planning problem $PP = \langle PD, q \rangle$ an optimistic plan exists is* PSPACE*-complete. Hardness holds even in the case where PP is plain (thus, proper).*

Proof. A proof of membership in PSPACE is the discussion above (note Lemma 1). We remark that the problem can be solved by a deterministic algorithm in polynomial workspace as follows. Similar as in [9], design a deterministic algorithm REACH(s, s', ℓ) which decides, given states s and s' and an integer ℓ, whether a sequence $t_1, \ldots, t_\ell$ of legal transitions $t_i = \langle s_{i-1}, A_i, s_i \rangle$ exists, where $s = s_0$ and $s' = s_\ell$, by cycling trough all states s'' and recursively solving REACH$(s, s'', \lfloor \ell \rfloor)$ and REACH$(s'', s', \lfloor \ell \rfloor + 1)$. Then, the existence of an optimistic plan of length ℓ can be decided cyclic through all pairs of states s, s' and testing whether s is a legal initial state, s'' satisfies the goal in given in q, and REACH(s, s', ℓ) returns true. Since the recursion depth is $O(\log \ell)$, and each level of the recursion needs only polynomial space, Lemma 1 implies that this algorithm runs in polynomial space.

For the PSPACE-hardness part, we describe how propositional STRIPS planning as in [9] can be reduced to planning in $\mathcal{K}$, where the planning domain PD is plain.

Recall that in propositional STRIPS, a state description s is a consistent set of ground literals, and an operator op has a precondition $pc(op)$, an add-list

$add(op)$, and a delete-list $del(op)$, which all are lists of propositional literals. The operator op can be applied in s if $pc(op) \subseteq s$ holds, and its execution yields the state $op(s) = (s \setminus del(op)) \cup add(op)$ (where s' must be consistent). Otherwise, the application of op on s is undefined. A goal γ, which is a set of literals, can be reached from a state s, if there exists a sequence of operators $op_1, \ldots, op_\ell$, where $\ell \geq 0$, such that $s_i = op_i(s_{i-1})$, for $i = 1, \ldots, \ell$, where $s_0 = s$, and $\gamma \subseteq s_\ell$ holds. Any such sequence is called a *STRIPS-plan* (of length ℓ) for s, γ. Given s, γ, a collection of STRIPS operators $op_1, \ldots, op_n$, and an integer $\ell \geq 0$, the problem of deciding whether some STRIPS-plan of length at most ℓ exists is PSPACE-complete [9]. As easily seen, this remains true if we ask for a plan of length exactly ℓ (just introduce a dummy operation with empty precondition and no effects).

Each STRIPS operator op_i is easily modeled as action in language $\mathcal{K}$ using the following statements in AD:

`executable` op_i `if` $pc(op_i)$	
`caused` L `after` op_i	for each $L \in add(op_i) \setminus del(op_i)$
`caused` L `after` op_i`,` L	for each $L \notin add(op_i) \cup del(op_i)$

The last rule is an inertia rule for the literals not affected by op.

The initial state s of a STRIPS planning problem can be easily represented using the following constraints in the `'initially'` section of a $\mathcal{K}$ program for PD:

`caused` L	for all $L \in s$.

Finally, AD contains the mandatory rules for unique action exection in a plain planning domain:

`caused false after` op_i`,` op_j	$1 \leq i < j \leq n$
`caused false after not` op_1`, not` op_2`, ..., not` op_n .	

It is easy to see that for the planning problem $PP = \langle PD, q \rangle$ where $q = \gamma\ ?\ (\ell)$, some optimistic plan exists iff a STRIPS-plan of length ℓ for s, γ exists. Since PP can be constructed in polynomial time from the STRIPS instance, this proves the PSPACE-hardness part. □

If the number of steps in q is fixed, the complexity decreases because altogether there is fixed number of guesses, which have polynomial size.

Theorem 2. *Deciding whether for a given ground planning problem $PP = \langle PD, q \rangle$ an optimistic plan exists is* NP*-complete, if the number of steps in q is fixed. Hardness holds even if PP is plain (thus, proper).*

Proof. The problem is in NP, since a legal transition sequence $T = \langle t_1, \ldots, t_i \rangle$ where $t_j = \langle s_{j-1}, A_j, s_j \rangle$ for $j = 1, \ldots, i$, such that s_i satisfies the goal in $q = Goal\ ?(i)$ can be guessed and verified in polynomial time if i is fixed.

NP-hardness for plain planning problems is shown by a simple reduction from the satisfiability problem (SAT). Let $\phi = C_1 \wedge \cdots \wedge C_k$ be a CNF, i.e., a

conjunction of clauses $C_i = L_{i,1} \vee \cdots \vee L_{i,m_i}$ where the $L_{i,j}$ are classical literals over propositional atoms $x_1, \ldots, x_n$.

Then, put the following clauses into the `'initially'` section:

`caused` x_j `if not` $-x_j$ for all $j = 1, \ldots, n$
`caused` $-x_j$ `if not` x_j for all $j = 1, \ldots, n$
`caused false if` $-L_{i,1}$, ..., $-L_{i,m_i}$ for all $i = 1, \ldots, k$
`caused` 0.

Here 0 is a further atom. Clearly, a legal initial state exists iff ϕ is satisfiable. Thus, a plan P exists for the query $Q = 0$? (0) iff ϕ is satisfiable. Since PD is easily constructed from ϕ, the result follows. □

Deciding the existence of a secure plan appears to be harder, since it allows us to encode also planning under incomplete initial states as in [2]. Already recognizing a secure plan is difficult.

Theorem 3. *Given a ground planning problem $PP = \langle PD, q \rangle$ and an optimistic plan P for PP, deciding whether P is secure is (a) Π_2^P-complete in general and (b) coNP-complete, if PP is proper. The hardness results hold even if the number of steps in q is fixed.*

Proof. The plan $P = \langle A_1, \ldots, A_i \rangle$ for PP is not secure, if a legal transition sequence $T = \langle t_1, \ldots, t_\ell \rangle$, where $t_j = \langle s_{j-1}, A_j, s_j \rangle$, for $j = 1, \ldots, \ell$ exists, such that either (i) $\ell = i$ and s_i does not satisfy the goal in q, or (ii) $\ell < i$ and for no state s, the tuple $\langle s_\ell, A_{\ell+1}, s \rangle$ is a legal transition. A legal transition sequence T of length ℓ can, by Corollary 1, be guessed and verified in polynomial time. Condition (i) can be easily checked. Condition (ii) can be checked by a call to an NP-oracle in polynomial time. It follows that checking security is in $\text{coNP}^{\text{NP}} = \Pi_2^P$ in general. If PP is proper, condition (ii) can be checked in polynomial time, and thus the problem is in coNP. This shows the membership parts.

Π_2^P-hardness in case (a) is shown by a reduction from deciding whether a quantified Boolean formula (QBF) $\Phi = \forall X \exists Y \phi$ is true, where X, Y are disjoint sets of variables and $\phi = C_1 \ldots C_k$ is a CNF over $X \cup Y$. This problem is well-known Π_2^P-complete, cf. [47]. Without loss of generality, we may assume that ϕ is satisfied if all atoms in $X \cup Y$ are set to true.

The `'initially'` section contains the following constraints:

`caused` x_j `if not` $-x_j$ for all $x_j \in X$
`caused` $-x_j$ `if not` x_j for all $x_j \in X$
`caused` 0

The `'always'` section contains the following rules. Suppose that $L_{i,1}, \ldots L_{i,n_i}$ are all literals over atoms from X that occur in C_i, and similarly that $K_{i,1}$, $\ldots K_{i,m_i}$ are all literals over atoms from Y that occur in C_i.

`caused` y_j `if not` $-y_j$ `after` 0 for all $y_j \in X$
`caused` $-y_j$ `if not` y_j `after` 0 for all $y_j \in X$
`caused false if` $-K_{i,1}, \ldots, -K_{i,m_i}$
`after` 0, $-L_{i,1}, \ldots, -L_{i,n_i}$ for all $i = 1, \ldots, k$
`caused` 1 `after` 0

Note that, by these rules, there are $2^{|X|}$ legal initial states $s_0^1, \ldots, s_0^{2^{|X|}}$ which correspond 1-1 to the truth assigments to the atoms in X. Each such s_0^i contains precisely one of x_j and $-x_j$, for all $x_j \in X$, and the atom 0. The causal rules for y_j and $-y_j$ effect that in each legal state s_1 wich follows the initial state, exactly one of y_j and $-y_j$ is contained. That is, s_1 must encode a truth assignment for Y. The rules with `false` in the head check that the assignment to $X \cup Y$, given jointly by s_0^i and s_1, must satisfy all clauses of ϕ. Furthermore, 1 must be contained in s_1 by the last rule.

Let us introduce a dummy action α. Then, the assumption on Φ implies that $P = \langle\langle s_0, A_1, s_1\rangle\rangle$, where $s_0 = X \cup \{0\}$, $A_1 = \{\alpha\}$, and $s_1 = X \cup Y \cup \{1\}$, is a legal transition sequence, and thus $P = \langle A_1\rangle$ is an optimistic plan for the query 1 ? (1).

It is not hard to see that P is secure iff Φ is true. Since $\langle PD, q\rangle$ is easily constructed from Φ, this proves the hardness part of (a). The hardness part of (b) is established by a variant of the reduction, in which Y is disregarded (i.e., $Y = \emptyset$), and the rules are modified as follows: `false` is replaced by 1, and the rule with effect 1 is dropped. Note that the resulting planning domain is plain. Then, the plan $P = \langle A_1\rangle$ is secure, iff $\forall X \neg\phi$ is true, i.e., the CNF ϕ is unsatisfiable. Since deciding this is coNP-complete, this proves coNP-hardness in case (b). □

When looking for a secure plan, the complexities of generating an optimistic plan and checking security combine, even if the number of steps is bounded. Intuitively, we can build a secure plan step by step only if we know all states that are reachable after the steps $A_1, \ldots, A_i$ so far when the next step A_{i+1} is generated. Either we store these states explicitly, which needs exponential space in general, or we store the steps $A_1, \ldots, A_i$ (from which these states can be recovered) which also needs exponential space in the representation size of $\langle PD, q\rangle$. In any case, such a nondeterministic algorithm for generating a secure plan needs exponential time.

In general, deciding whether a given ground planning problem $PP = \langle PD, q\rangle$ has a secure plan is NEXPTIME-complete. Hardness holds even if the planning problem is plain (thus, proper), this result is given without proof here (proof can be found in [17]. Hence, NEXPTIME actually captures the complexity of deciding the existence of a secure plan. Note that NEXPTIME strictly contains PSPACE, and thus this problem cannot be efficiently translated to traditional STRIPS planning.

More interesting for our observations in the following is the existence of a secure plan if the number of steps is fixed:

Theorem 4. *Deciding whether a given ground planning problem $PP = \langle PD, q\rangle$ has a secure plan is (a) Σ_3^P-complete if the number of steps in q is fixed and (b) Σ_2^P-complete if number of steps in q is fixed and PP is proper (Σ_2^P-hardness holds even for plain PP).*

Proof. A legal transition sequence $T = \langle\langle s_0, A_1, s_1\rangle, \ldots, \langle s_{i-1}, A_i, s_i\rangle\rangle$ of fixed length i that induces an optimistic plan $P = \langle A_1, \ldots, A_i\rangle$ can be guessed and verified in polynomial time (Corollary 1), and by Theorem 3, checking whether P is secure is possible with a call to an oracle for Π_2^P in case (a) and for coNP in case (b). Hence, it follows that the problem is in Σ_3^P in case (a) and in Σ_2^P in case (b).

For the hardness part of (a), we transform deciding the validity of a QBF $\Phi = \exists Z \forall X \exists Y \phi$, where X, Y, Z are disjoint sets of variables and $\phi = C_1 \ldots C_k$ is a CNF over $X \cup Y \cup Z$, which is Σ_3^P-complete [47], into this problem. The transformation is in spirit of the reduction in the proof of Theorem 3.

The `'initially'` section of the $\mathcal{K}$ program for PD contains the following constraints:

`caused` x_j `if not` $-x_j$ for all $x_j \in X$
`caused` $-x_j$ `if not` x_j for all $x_j \in X$
`caused 0.`

We introduce, for each atom $z_i \in Z$, an action $\texttt{set}_{z_i}$, which has the following executability condition:

`executable` $\texttt{set}_{z_i}$ `if 0.`

The `'always'` section of the program contains the following rules. Suppose that $L_{i,1}, \ldots L_{i,n_i}$ are all literals over atoms from $X \cup Z$ that occur in C_i, and similarly that $K_{i,1}, \ldots K_{i,m_i}$ are all literals over atoms from Y that occur in C_i.

`caused` x_j `after 0,` x_j for all $x_j \in X$
`caused` $-x_j$ `after 0,` $-x_j$ for all $x_j \in X$
`caused` z_i `after 0,` $\texttt{set}_{z_i}$ for all $z_i \in Z$
`caused` $-z_i$ `after 0, not` $\texttt{set}_{z_i}$ for all $z_i \in Z$
`caused 1 after 0`
`caused` y_j `if not` $-y_j$ `after 1` for all $y_j \in Y$
`caused` $-y_j$ `if not` y_j `after 1` for all $y_j \in Y$
`caused false if` $-L_{i,1}, \ldots, -L_{i,n_i}$
`after 1,` $-K_{i,1}, \ldots,$ K_{i,m_i} for all $i = 1, \ldots, k$
`caused 2 after 1.`

Given these action descriptions, there are $2^{|X|}$ many legal initial states s_0^1, $\ldots$, $s_0^{2^{|X|}}$, which correspond 1-1 to the possible truth assignments to the variables in X and contain 0. Only in these states s_0^i actions are executable, which assign a subset of Z the value true. Every state s_1^i reached from s_0^i by a legal transition must, for each atom $\alpha \in Z \cup X$, either contain α or $-\alpha$, and it

contains the atom 1. Next, each state s_2 reached from such a s_1^i must contain either y_j or $-y_j$, for every $y_j \in Y$, and it contains the atom 2.

It is not hard to see that an optimistic plan for the goal 2 exists iff there is an assignment to all variables in $X \cup Y \cup Z$ such that the formula ϕ is satisfied. Any such plan must be of form $P = \langle A_1, \emptyset \rangle$, where $A_1 \subseteq \{\mathtt{set}_{z_i} \mid z_i \in Z\}$. Furthermore, P is secure iff A_1 represents an assignment to the variables in Z such that, regardless of which assignment to the variables in X is chosen (which corresponds to the legal initial states s_0^i), there is some assignment to the variables in Y (i.e., there is at least some state s_2^i reachable from s_1^i, by doing nothing), such that all clauses of ϕ are satisfied; any such s_2^i contains 2. In other words, P is secure iff Φ is true.

Since *PD* is constructible from Φ in polynomial time, it follows that deciding whether a secure plan exists for $PP = \langle PD, q \rangle$, where $q = 2$? (2), is Σ_3^P-hard. This proves part (a).

For the hardness part of (b), we modify and simplify the construction for part (a) as follows. We assume that $Y = \emptyset$, and

- replace `false` in rule heads by 1, and 1 in the bodies by 0.
- remove the rules for 1 and 2 (and those for y_j, $-y_j$).

Clearly, the resulting planning domain is plain. We have again $2^{|X|}$ initial states s_0^i, which correspond to the truth assignments to X. An optimistic plan for the goal 1 is of the form $P = \langle A_1 \rangle$ where $A_1 \subseteq \{\mathtt{set}_{z_i} \mid z_i \in Z\}$ and corresponds to an assignment to $Z \cup X$ such that ϕ is *false*. The plan P is secure iff every assignment to X, extended by the assignment to Z encoded by A_1, makes ϕ false.

It follows that a secure plan for $PP = \langle PD, q \rangle$, where $q = 1$? (1), exists iff the QBF $\exists Z \forall X \neg \phi$ is true. Evaluating a QBF of this form is Σ_2^P-hard (recall that ϕ is in CNF). Since PP is constructible in polynomial time, this proves Σ_2^P-hardness for part (b). □

If we exclude concurrent actions, the complexity remains unaffected in the general case (cf. Theorem 4). However, it is lower if the plan length is fixed. Recall that D^P is the class of problems which are the conjunction of a problem in NP and a problem in coNP.

Theorem 5. *Deciding whether a given ground planning problem $PP = \langle PD, q \rangle$ has a secure sequential plan is (a) Π_2^P-complete, if the number of steps in q is fixed, and (b) D^P-complete if the number of steps in q is fixed and PP is proper (D^P-hardness holds even for plain PP).*

Proof. If the plan length i in the query $q =$ *Goal* ? (i) is fixed, the number of candidate sequential secure plans, given by $(a+1)^i$, where a is the number of actions in *PD*, is bounded by a polynomial.

A candidate $P = \langle A_1, \ldots, A_n \rangle$ is not a secure plan, if (i) no initial state s_0 exists, or (ii) like in the proof of Theorem 3, a legal transition sequence $T = \langle t_1, \ldots, t_\ell \rangle$, where $t_j = \langle s_{j-1}, A_j, s_j \rangle$, for $j = 1, \ldots, \ell$ exists, such that

either (ii.1) $\ell = i$ and s_i does not satisfy the goal in q, or (ii.2) $\ell < i$ and for no state s, the tuple $\langle s_\ell, A_{\ell+1}, s\rangle$ is a legal transition. The test for (i) is in coNP, while the test for (ii) is in Σ_2^P in general and in NP if PP is proper (cf. proof of Theorem 3). Note that (i) is identical for all candidates.

Thus, the existence of a sequential secure plan can be decided by the conjunction of a problem in NP and a disjunction of polynomially many instances of a problem in Π_2^P in case (a) and in coNP in case (b); since NP $\subseteq \Pi_2^P$ and both Π_2^P and coNP are closed under polynomial disjunctions and conjunctions of instances (i.e., a disjunction resp. conjunction of instances can be polynomially transformed into a single instance), it follows that the problem is in Π_2^P in case (a) and in D^P in case (b).

Π_2^P-hardness for case (a) follows from the reduction in the proof of Theorem 3. There, a secure, sequential plan exists for the query 1 ? (1) iff the plan $P = \langle\{\alpha\}\rangle$ is the secure.

D^P-hardness in case (b) is shown by a reduction from deciding, given CNFs $\phi = \bigwedge_{i=1}^{n} L_{i,1} \vee L_{i,2} \vee L_{i,3}$ and $\psi = \bigwedge_{j=1}^{m} K_{j,1} \vee K_{j,2} \vee K_{j,3}$ over disjoint sets of atoms X and Y, respectively, whether ϕ is satisfiable and ψ is unsatisfiable.

The `'initially'` section of the $\mathcal{K}$-program for PD contains the following constraints:

`caused` x_j `if not` $-x_j$ for all $x_j \in X$
`caused` $-x_j$ `if not` x_j for all $x_j \in X$
`caused` y_j `if not` $-y_j$ for all $y_j \in Y$
`caused` $-y_j$ `if not` y_j for all $y_j \in Y$
`caused` $L_{i,1}$ `if` $-L_{i,2}$`,` $-L_{i,3}$ for all $i = 1, \ldots, n$

The `'always'` section contains the following rules:

`caused` f `after` $-K_{i,1}$ $-K_{i,2}$`,` $-K_{i,3}$ for all $i = 1, \ldots, m$

Obviously, these rules satisfy the conditions for a plain planning domain. Now introduce a dummy action α, and add the mandatory action execution constraints of a plain planning domain (which enforce that α must be executed). Then, for the query $q = f$? (1), the only candidate for a sequential secure plan is $P = \langle\{\alpha\}\rangle$. As easily seen, P is a secure plan for q iff ϕ is satisfiable (which is equivalent to the existence of legal initial states) and ψ is unsatisfiable (which means that f is true in each state reachable by executing α on an initial state). This proves the hardness part of (b). □

3.3.2 Conclusions and Considerations upon Implementation

The Σ_3^P- and Σ_2^P-completeness results in Theorem 4 imply that even short secure plans can not be efficiently expressed in systems which allow to solve only problems in NP, such as Blackbox [34], CCALC [39], smodels [45], or satisfiability checkers. The hardness results rely on the fact that parallel actions are possible. Note that Baral et al. [2] report the related result that deciding

in language $\mathcal{A}$ [27], which gives rise to proper, deterministic planning domains, the existence of an, in our terminology, secure sequential plan whose length is polynomially bounded is Σ_2^P-complete.

Note that furthermore D^P-completeness in Theorem 5 implies that the secure planning problem is still not efficiently representable in systems with expressiveness limited to NP. Nevertheless we will present a translation to the more powerful formalism of Answer Set Programming, i.e. logic programming with classical negation and disjunction in the head, also referred as Extended Disjuncitve Logic Programming. EDLP under the stable model semantics is Π_2^P-complete for the propositional case due to [13], [12].

This allows the following conclusions considering the translation to ASP:

Theorem 2 shows that optimistic planning with fixed plan length can be reduced efficiently to extended LP under the answer set semantics, which will be shown in the next section, as both problems are NP-complete: Given a propositional extended logic program, deciding whether an answer set exists is NP-complete [4]. Properness does not matter for optimistic planning.

By a related result in [4], extended logic programming (logic programming with classical negation) is coNP-complete. This means that there must be an efficient algorithm in extended logic programming to check plan security for proper domains due to the result in Theorem 3(b).

Unfortunately, checking whether a planning domain is proper is not feasible in general, so we will reduce our observations to a certain subclass of proper domains. As stated above, plain domains remain proper, if we extend them by stratified default negation. The translation, which will be proposed subsequently in fact only works correctly on stratified domains, although due to the results above there should be a translation of the check to extended logic programming which works for all proper domains.

At least it would be interesting to lift the restriction to stratified domains a little bit more by finding a more general class of proper domains, which are easy to check. Stratification restricts us to programs that for example do not even allow `total` statements or pairs of `inertial` positive and negative rules:

```
caused f if not -f after f.
caused -f if not f after f.
```

Obviously such a pair of rules standing alone does not harm properness, as they never can fire concurrently.

There exist promising results in [48] which state that we could possibly allow default negation more generously while properness of domains is still guaranteed, this will be part of further investigations.

The further results finally allow the assumption and hope that even an efficient translation for general secure checking (Theorem 3(a)) and the problem of secure plan existence for sequential planning (Theorem 5) can be found using the increased expressive power of **disjunctive** logic programming, which is also Π_2^P-complete as stated above. Also the Σ_2^P result in Theorem 4 seems to be

very promising for finding further translations. This will also be part of further investigations.

Chapter 4

From $\mathcal{K}$ to Disjunctive Logic Programming

In order to find plans for problems given in language $\mathcal{K}$, it is desirable to translate these planning problems to problems in logic programming which can be processed by logic programming systems, like DLV [23] or Smodels [46, 52]. These systems can efficiently find models under the answer set semantics and stable model semantics respectively.

The translation given in here is based on the translation for Action Language $\mathcal{C}$ proposed by Lifschitz, see Section 2.2, but is closer to logic programming: Our language for example allows default negation directly and the translation will be straightforward and very simple.

After giving a basic translation, which allows us to find optimistic plans for a given planning problem by calculating the answer sets of a logic program, some refinements will be explained and a method for checking plan security using logic programming will be shown, which is applicable for a certain subclass of (proper) planning domains.

4.1 Basic Translation

Given a (non-ground) $\mathcal{K}$ planning problem $P = \langle PD, q\rangle$ we will now give a simple translation that will create a disjunctive logic program P^{LP}. It will be shown that the answer sets of this logic program correspond with the valid (optimistic) plans of P.

We start with the translation of the query q:

$$g_1(\overline{X}_1), \ldots, g_m(\overline{X}_m), \texttt{not } g_{m+1}(\overline{X}_{m+1}), \ldots, \texttt{not } g_n(\overline{X}_n)? \quad \texttt{(i)}$$

is translated to:

$$\begin{array}{l} time(T) \ :- \ 0 \leq T \leq i. \\ actiontime(T) \ :- \ 0 \leq T < i. \\ goal \ :- g_1(\overline{X}_1, i), \ldots, g_m(\overline{X}_m, i), \texttt{not } g_{m+1}(\overline{X}_{m+1}, i), \ldots, \texttt{not } g_n(\overline{X}_n, i). \\ :- \texttt{not } goal. \end{array} \tag{4.1}$$

This translation introduces two auxiliary predicates *time*, *actiontime*, which will be used to describe time as a parameter for the actions and fluents. $time(T)$ is true for all possible points of time $T \in \{0, 1, \ldots, \texttt{i}\}$, whereas $actiontime(T)$ is true only for all but the maximum points of time, to describe that actions may not occur in the final state.

4.1.1 Typing and Safety

For $PD = \langle \Pi, AD \rangle$ the background knowledge Π is already given as a logic program and all the rules in Π and facts can be directly included into P^{LP} without translation.

The initial state constraints, causation rules and executability conditions in AD then have to be transformed according to the rewriting pointed out in Section 3.1.7 to achieve safety in the translated program.

The resulting planning domain will be translated as follows:

4.1.2 Executability Conditions

Each executability condition e:
`executable` $a(\overline{X})$ `if` $b_1(\overline{X}_1), \ldots, b_m(\overline{X}_m), \texttt{not } b_{m+1}(\overline{X}_{m+1}), \ldots, \texttt{not } b_n(\overline{X}_n)$
is translated to the following LP rule:

$$\begin{array}{l} a(\overline{X}, T) \ \vee \ \neg a(\overline{X}, T) \ :- \ actiontime(T), b_1(\overline{X}'_1), \ldots, b_m(\overline{X}'_m), \\ \qquad\qquad \texttt{not } b_{m+1}(\overline{X}'_{m+1}), \ldots, \texttt{not } b_n(\overline{X}'_n). \end{array} \tag{4.2}$$

where $b_i(\overline{X}'_i)$ $(i \in \{1, \ldots, n\})$ denotes the literal

- $b_i(\overline{X}_i)$ iff $b_i \in \sigma^{typ}$ and
- $b_i(\overline{X}_i, T)$ iff $b_i \in \sigma^{fl}$.

These rules "guess", whether an action occurs at a certain point of time or not, for example given that action `move` has the declaration:
`move(B,L) requires block(B), location(L).`
we translate
`executable move(B,L) if not occupied(B), not occupied(L), B` $\neq$ `L.`
to:

```
move(B,L,T) v -move(B,L,T) :- block(B), location(L), not occupied(B),
                              not occupied(L), B ≠ L.
```

4.1.3 Causation Rules

Each causation rule c:

$$\texttt{caused } f(\overline{X}) \texttt{ if } b_1(\overline{X}_1), \ldots, b_k(\overline{X}_k), \texttt{not } b_{k+1}(\overline{X}_{k+1}), \ldots, \texttt{not } b_l(\overline{X}_l)$$
$$\texttt{after } a_1(\overline{Y}_1), \ldots, a_m(\overline{Y}_m), \texttt{not } a_{m+1}(\overline{Y}_{m+1}), \ldots, \texttt{not } a_n(\overline{Y}_n)$$

is translated to the following LP rule:

$$\begin{aligned} f(\overline{X}, T1) \ :- \ & b_1(\overline{X}'_1), \ldots, b_k(\overline{X}'_k), \texttt{not } b_{k+1}(\overline{X}'_{k+1}), \ldots, \texttt{not } b_l(\overline{X}'_l), \\ & a_1(\overline{X}'_1), \ldots, a_m(\overline{X}'_m), \texttt{not } a_{m+1}(\overline{Y}'_{m+1}), \ldots, \texttt{not } a_n(\overline{Y}'_n), \\ & \mathit{actiontime}(T), \ T1 = T + 1. \end{aligned} \tag{4.3}$$

where $b_i(\overline{X}'_i)$ $(i \in \{1, \ldots, l\})$ denotes the literal

- $b_i(\overline{X}_i)$ iff $b_i \in \sigma^{typ}$ and
- $b_i(\overline{X}_i, T1)$ iff $b_i \in \sigma^{fl}$.

and $a_j(\overline{Y}'_j)$ $(j \in \{1, \ldots, n\})$ denotes the literal

- $a_j(\overline{Y}_j)$ iff $a_j \in \sigma^{typ}$ and
- $a_j(\overline{Y}_j, T)$ iff $a_j \in \sigma^{act} \cup \sigma^{fl}$.

For rules with an empty `after` part (static rules), $T1 = T + 1$ may be skipped in the resulting LP rules, as the variable T does not appear anywhere else in the rule.

A short example (given fluent `on` has the declaration: `on(B) requires block(B).` and move has the same declaration as mentioned above)

```
caused on(B,L) after move(B,L).
caused -on(B,L1) after move(B,L), on(B,L1), L ≠ L1.
```

will be translated to:

```
on(B,L,T1)    :- move(B,L,T), block(B), location(L), actiontime(T),
                 T1 = T + 1.
-on(B,L1,T1) :- move(B,L,T), block(B), location(L1), on(B,L1),
                 actiontime(T), T1 = T+1.
```

For rules having `false` in the `caused` part and an empty `if` part (for example the `nonexecutable` rules) $\mathit{actiontime}(T)$, $T1 = T + 1$ can be replaced with the single literal $\mathit{actiontime}(T)$, whereas on the other hand for rules having `false` in the `caused` part and an empty `after` part $\mathit{time}(T1)$ is sufficient (as operator T will not occur in the translation.)

So rules with `false` in the `caused` part are translated to LP rules with an empty head (often denoted as (strong) integrity constraints in the literature [7], [6], [21], [8]), for instance

```
caused false if on(B,L), on(B,L1), L<>L1.
```

is translated to:

```
:- on(B,L,T1), on(B,L1,T1), L<>L1, time(T1).
```

4.1.4 Initial State Constraints

Finally each initial state constraint *ic*:
`initially caused` $f(\overline{X})$ `if` $b_1(\overline{X_1}), \ldots, b_m, (\overline{X_m}), \texttt{not}\ b_{m+1}(\overline{X_{m+1}}), \ldots, \texttt{not}\ b_n(\overline{X_n})$.
is translated to:

$$f(\overline{X}, 0)\ :-\ b_1(\overline{X}'_1), \ldots, b_m(\overline{X}'_m), \texttt{not}\ b_{m+1}(\overline{X}'_{m+1}), \ldots, \texttt{not}\ b_n(\overline{X}'_n). \quad (4.4)$$

where $b_i(\overline{X}'_i)$ $(i \in \{1, \ldots, n\})$ denotes the literal

- $b_i(\overline{X}_i)$ iff $b_i \in \sigma^{typ}$ and
- $b_i(\overline{X}_i, 0)$ iff $b_i \in \sigma^{fl}$.

Example:
`initially caused false if on(B,L), on(B,L1), L` $\neq$ `L1.`
is translated to:
`:- on(B,L,0), on(B,L1,0), L` $\neq$ `L1.`

The inequality predicate "$\neq$" and the less predicate "$<$" are assumed to have the normal meaning.

4.1.5 Compact Translation for the `total` Statement

It is obvious that the basic translation for the `total` statement is rather inefficient: A statement

`total` f `if` $b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_l$
`after` $a_{l+1}, \ldots, a_m, \texttt{not}\ a_{m+1}, \ldots, \texttt{not}\ a_n$.

is first rewritten to two causation rules

`caused` f `if not` $\neg f, b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_l$
`after` $a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n$.
`caused` $\neg f$ `if not` $f, b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_l$
`after` $a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n$.

which are then translated to two LP rules following the basic translation 4.3. It is easy to see that in disjunctive logic programming this can be modeled easier using a guessing rule similar to the translation 4.2 for the executability of actions. So we directly translate the rule:

Each totalization rule t is first transformed according to the rewriting proposed in section 3.1.7 to achieve safety. The resulting executability condition t':

`total` f `if` $b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_l$
`after` $a_1, \ldots, a_m, \texttt{not}\ a_{m+1}, \ldots, \texttt{not}\ a_n$.

is then translated to the following LP rule:

$$f(\overline{X},T1) \ \vee \ \neg f(\overline{X},T1) \ :- \ b_1(\overline{X}'_1),\ldots,b_k(\overline{X}'_k),\texttt{not}\ b_{k+1}(\overline{X}'_{k+1}),\ldots,\texttt{not}\ b_l(\overline{X}'_l),\\ a_1(\overline{X}'_1),\ldots,a_m(\overline{X}'_m),\texttt{not}\ a_{m+1}(\overline{Y}'_{m+1}),\ldots,\texttt{not}\ a_n(\overline{Y}'_n),\\ \mathit{actiontime}(T),\ T1 = T+1. \tag{4.5}$$

where $b_i(\overline{X}'_i)$ $(i \in \{1,\ldots,l\})$ denotes the literal

- $b_i(\overline{X}_i)$ iff $b_i \in \sigma^{typ}$ and
- $b_i(\overline{X}_i,T1)$ iff $b_i \in \sigma^{fl}$.

and $a_j(\overline{Y}'_j)$ $(j \in \{1,\ldots,n\})$ denotes the literal

- $a_j(\overline{Y}_j)$ iff $a_j \in \sigma^{typ}$ and
- $a_j(\overline{Y}_j,T)$ iff $a_j \in \sigma^{act} \cup \sigma^{fl}$.

These rules "guess" the truth value of a fluent in a each state. When the totalization rule is under t' initial scope (i.e. preceded with the keyword `initially`), we can proceed analogously, skipping the $\mathit{actiontime}(T)$ and $T1 = T+1$ and substitute all occurrences of the time variable T in rule (4.5) with 0.

4.1.6 A Short Example

As a short example we will give the Axel Shooting domain already mentioned in Section (2.1.3):

Background knowledge is empty for this domain and there is one fluent *alive* and one action *shoot* which take one argument, *shoot* is always executable and can kill a person, so PD consists of the following (safe) rules and declarations:

$\mathit{shoot}(X)$ `requires` $\mathit{person}(X)$.
$\mathit{alive}(X)$ `requires` $\mathit{person}(X)$.

`executable` $\mathit{shoot}(X)$.
`initially caused` $\mathit{alive}(\mathit{axel})$.
`inertial` $\mathit{alive}(X)$ `if` $\mathit{person}(X)$.
`total` $\mathit{alive}(X)$ `after` $\mathit{shoot}(X)$, $\mathit{person}(X)$.

The goal query q is of course that Axel survives:

alive(axel)? (1)

The background knowledge contains one fact:

$\mathit{person}(\mathit{axel})$.

The translated program P^{LP} is shown be in Figure 4.1. It has two models

$m_1 = \{\ person(axel), time(0), time(1), actiontime(0), alive(axel, 0),$
$\quad -shoot(axel, 0), alive(axel, 1), goal\}$
$m_2 = \{\ person(axel), time(0), time(1), actiontime(0), alive(axel, 0),$
$\quad shoot(axel, 0), alive(axel, 1), goal\}$

resp. optimistic plans: $p_1 = \langle\emptyset\rangle$, $p_2 = \langle\{shoot(axel)\}\rangle$, that means either no action is performed and *axel* survives or *axel* survives the shot.

4.1.7 Soundness and Completeness of the Basic Translation

The answer sets of the generated LP program P^{LP} obviously correspond to the optimistic plans of the original $\mathcal{K}$ program:

Theorem 6. *Given a (non-ground) $\mathcal{K}$ planning problem $P = \langle PD, q\rangle$ each answer set of P^{LP} corresponds to an optimistic plan of P (***soundness of Translation***) and each optimistic plan has some corresponding stable model of P^{LP} (***completeness of translation***).*

We have seen that in the translation all action and fluent literals are extended by a time stamp. Given an answer set M_P of P^{LP} we get the corresponding optimistic plan by ordering the action literals in the model by this time stamp: For all fluent literals in M_P with time stamp i corresponds with the state, which the literal belongs to and all action literals in the answer set with a common time stamp correspond to an executable action set. Together these build up a legal transition sequence which makes the goal true and therefore an optimistic plan.

First of all it is obvious by the translation that the grounding of ${P'}^{LP}$ of P^{LP} by definition exactly represents the translation of the $P' = \langle PD', q\rangle$ where PD' is the **legal instantiation** of PD, which means we can focus on ground planning problems in the following.

Another obvious observation is that $M \subseteq M_P$. This means that the model of the static background knowledge is included in each answer set of ${P'}^{LP}$.

In the following it will outlined how to prove that each optimistic plan of P has a corresponding answer set of P^{LP} and vice versa:

Proof sketch: Soundness. The proof sketch is based on the following lemma:

Lemma 2. *If M_P is an answer set of P^{LP} then the set of all positive and negative fluent literals with time stamp 0 $s_0 \subseteq M_P$ represents a legal initial state given by the set of fluent literals s_0' which is defined by deleting the timestamp operator from the literals in s_0.*

> *Proof sketch: Lemma.* To prove this we, first have to show that the corresponding fluents build a legal initial state:
>
> By definition of the translation all the fluents in s_0 can only be inferred by the translation of initial state constraints and static rules

$time(T) \ :- \ 0 \leq T \leq i.$
$actiontime(T) \ :- \ 0 \leq T < i.$
$goal \ :- \ alive(axel, 1).$
$:- \ \texttt{not} \ goal.$

$shoot(X,T) \vee \neg shoot(X,T) \ :- \ actiontime(T), person(X).$
$alive(axel, 0) :- .$
$alive(X,T1) \ :- \ \texttt{not} \ \neg alive(X,T1), alive(X,T), person(X),$
$\qquad actiontime(T), T1 = T + 1.$
$alive(X,T1) \vee \neg alive(X,T1) \ :- \ shoot(X,T), person(X),$
$\qquad actiontime(T), T1 = T + 1.$

Figure 4.1: Translation of a simple planning domain to a logic program

(with timestamp 0): For programs with default negation it's easy to see that the reduction P^t with $t = \langle \emptyset, \emptyset, s'_0 \rangle$ exactly coincides with the disjunctive logic program $P^{LP^{M_P}}$, concerning the initial state constraints and static rules (with timestamp 0). So if M_P ($\supseteq s_0$) is an answer set, then it also follows that s'_0 is a legal initial state, which finally proofs the lemma. □

In a similar way it can be shown that the set of all action atoms A_i with timestamp i in the answer set M_P ($a(i) \in M_P$) represent an executable action set for state i represented by s_i, the set of all fluent literals with time step i, and subsequently that $\langle s'_i, A'_i, s'_{i+1} \rangle$ is a legal state transition (s'_i, A'_i, s'_{i+1} are the corresponding fluent literals/action atoms built from $s_i, A_i, s_{i+1} \subseteq M_P$ by removing the time stamp).

Furthermore, the constraint 4.1 guarantees that the goal query is satisfied in the last state, and this finally shows part one (soundness) of theorem 6. □

Proof sketch: Completeness. Assume $\langle A_0, \ldots, A_{i-1} \rangle, i \geq 0$ is an optimistic plan. From 3.2.5 we know that consequently a legal transition sequence $T = \langle \langle s_0, A_0, s_1 \rangle, \ldots, \langle s_{i-1}, A_{i-1}, s_i \rangle \rangle$ in *PD* exists such that T establishes the goal.

Proof idea: Now, if we add the time stamps to the corresponding fluent/action literals, we have to show that:

$$M \cup s'_0 \cup A'_0 \cup s'_1 \cup \ldots \cup s'_{i-1} \cup A'_{i-1} \cup s'_i \cup \neg\overline{A'_0} \cup \ldots \cup \neg\overline{A'_{i-1}} \cup$$
$$\{time(0), actiontime(0), \ldots, actiontime(i-1), time(i-1), time(i), goal\}$$

is an answer set of P^{LP}, where s'_k (A'_k resp.) are the literals (atoms) obtained from extending fluent literals s_k (action atoms A_k) by time stamp k and $\neg\overline{A'_k}$ is the set of all the negative literals obtained from (i) building the set $\overline{A_k}$ of all legal action instantiations $\notin A_k$ (ii) extending the action atoms in this set by time stamp k, (iii) classically negating all the atoms in this set $\overline{A'_k}$.

It follows by construction of the translated rules and constraints (4.1)-(4.4) and the definition of the language that the set constructed above is an answer set for P^{LP}. □

4.2 Extending the Translation

The basic translation is sufficient for translating optimistic planning of $\mathcal{K}$ programs. Also the enhanced statements `inertial`, `default`, `nonexecutable` and `total` can be translated straightforward using the basic translation for their corresponding static and dynamic rules. Still we cannot express sequential plans (see Section 3.2.6) and cannot describe secure plans (see Section 3.2.5) in terms of logic programming.

In this section a solution for these two problems will be given by extending the basic translation. Furthermore, at the end of this section we will propose an optimization to make the translation more efficient.

4.2.1 How to Rule Out Concurrent Actions

In order to allow sequential plans only (see Section 3.2.6) we have to add constraints to P^{LP} which forbid all concurrent actions. For all action names a occurring in PD we add:

$$
\begin{aligned}
&:- a(X_1, X_2, \ldots, X_n, T), a(Y_1, Y_2, \ldots, Y_n, T), X_1 \neq Y_1.\\
&:- a(X_1, X_2 \ldots, X_n, T), a(Y_1, Y_2, \ldots, Y_n, T), X_2 \neq Y_2.\\
&\vdots\\
&:- a(X_1, X_2, \ldots, X_n, T), a(Y_1, Y_2, \ldots, Y_n, T), X_n \neq Y_n.
\end{aligned}
$$

where n is the arity of action a. These constraints rule out all concurrent actions of the same action name. Further more we have to forbid all different actions to occur at once, so add a constraint

$$:- a(X_1, X_2, \ldots, X_m, T), a'(Y_1, Y_2, \ldots, Y_n, T).$$

for each pair of different action names in a, a' in PD, where m denotes the arity of action a and n the arity of a' respectively.

For example in a blocks world domain, where there is only one action `move(B,L)`, expressing a block `B` is moved to a location `L`, we add the following constraints to P^LP in order to achieve sequential plans:

$$
\begin{aligned}
&\texttt{:- move}(X_1, X_2, T)\texttt{, move}(Y_1, Y_2, T)\texttt{,}\ X_1 \neq Y_1.\\
&\texttt{:- move}(X_1, X_2, T)\texttt{, move}(Y_1, Y_2, T)\texttt{,}\ X_1 \neq Y_1.
\end{aligned}
$$

4.2.2 Security Check for Plans

Due to the complexity results in the previous section 3.3 it is obvious that secure (conformant) planning cannot simply be reduced to answer set programming in general. Still it is possible to give a translation to check a plan for security, which works for a subclass of proper domains, which will be presented now.

Checking Properness of Domains

According to the complexity results in the previous chapter (Section 3.3) we saw that in general checking security is rather difficult, even with fixed plan length, though the complexity of the problem is much lower for proper planning domains, allowing us to translate the plan check to Extended Logic Programming.

Checking properness on the other hand is not a trivial problem as well, so it is best to restrict ourselves to a certain subclass of programs, which are proper for sure.

A simple method is to only allow plain domains, which on the other hand reduces the expressive power of general $\mathcal{K}$ and the possibilities to express many practical examples. Especially it is impossible to deal with uncertainty of any kind apart from the initial state. A possible extension to this is additionally allowing default negation as long as it remains stratified due to Definition 25.

Unfortunately allowing only stratified negation in the *post* part of all causation rules under always scope is rather restrictive, as, for example, `total` rules or a pair of rules like:

`caused` f `if not` $-f$ `after` f.
`caused` $-f$ `if not` f `after` $-f$.

will be forbidden. But these rules simply represent inertia of f and $-f$, which standing alone does not harm properness as f and $-f$ are mutually exclusive. General use of `total` at least could be emulated by using the approach shown in Section 3.3 for fixed plan length, however,

Some more sophisticated ideas to allow a more generous use of default negation will be mentioned in the prospects given in Chapter 8, the current frontend implements only a very simple stratification check on the non-ground program, described in Chapter 5.

Security Check for Stratified Domains

The approach shown here is strongly related to cautious reasoning in logic programming ([18], [14]). The basic idea how to realize cautious query evaluation in DLV depends on the following assumption: When we want to know whether a goal is bravely true (i.e. there exists a model which contains the goal) we add a constant

$:-$ `not` *goal*.

which rules out all models not containing the goal. This more or less coincides with the translation of planning goals in P^{LP} (see 4.1).

On the other hand, if we want to guarantee that the goal is true in all models, we have to turn the question round and show that there is no model **NOT** containing the goal, so we modify the constraint to:

$$:- \mathit{goal}.$$

and whenever we find no model, then "goal" is cautiously true. Unfortunately this is not sufficient for our translation of planning problems to implement secure planning, the main difference is that here we also have to guarantee plan execution to be checked apart from the goal itself:

When a plan is p secure that means that

- the goal is true in all models (this is more or less the cautious reasoning part) **AND**
- all actions of plan p are executable under any possible initial state and
- the execution of p must be guaranteed, and no other actions are allowed to be executed.
- any execution of an action of p leads to a legal subsequent state

Thus, rewriting the goal alone is not sufficient. Finding secure plans takes two steps:

1. Finding an optimistic plan p
2. rewriting the program P^{LP}, such that the restrictions above are fulfilled.

Model generation for a non-ground logic program like P^{LP} is divided in two steps:

1. grounding the program.
2. model generation.

As for large program instances grounding is an essential task in terms of performance, it would be desirable to find a way to check plan security in terms of the grounded program by adding constraints and rules to the ground version ${P^{LP}}_{\downarrow}$.

The models generated then correspond to optimistic plans, as shown above. To check whether an optimistic plan of length n is secure we first add the plan (which is of course ground by definition) $p = \langle A_0, A_1, \ldots, A_{n-1} \rangle$, where

$A_i = \{a_{i,1}(\overline{X_{i,1}}), \ldots, a_{i,k_i}(\overline{X_{i,k_i}})\}$ to the ground program $P^{LP}{\downarrow}$ as constraints which forbid that the actions of the plan do NOT occur:

$$
\begin{array}{l}
:- \neg a_{0,1}(\overline{X_{0,1}}, 0). \\
\vdots \\
:- \neg a_{0,k_0}(\overline{X_{0,k_0}}, 0). \\
\\
\vdots \\
\\
:- \neg a_{n-1,1}(\overline{X_{0,1}}, n-1). \\
\vdots \\
:- \neg a_{n-1,k_{n-1}}(\overline{X_{n-1,k_{n-1}}}, n-1).
\end{array}
\qquad (4.6)
$$

This forces the guessing rules generated in the basic translation to allow only guesses where actions of the checked plan p occur. Furthermore, we have to remove all guessing rules from $P^{LP}{\downarrow}$ which do not guess actions in the plan p: remove all ground rules of the form 4.2:
$a_t(t) \vee \neg a_t(t) :- actiontime(t), B_1, \ldots, B_m, \texttt{not } B_{m+1}, \ldots, \texttt{not } B_n.$
where $a_t \notin p$. This guarantees that no other actions than the actions in the checked plan can occur.

Now we have to change the *goal* constraint to enforce cautious reasoning. This could be managed by simply rewriting the goal constraint:

$$:- \texttt{not } goal.$$

to

$$:- goal.$$

Whenever a model is found, we can conclude that there is a possible situation when the goal is not achieved, i.e. the goal query is cautiously true when no model is found for the modified program.

Still, rewriting the goal constraint alone still does not guarantee plan security:

1. situations when actions in the plan p are not `executable` are not covered, as in this case neither $a_t(t)$ nor $\neg a_t(t)$ are in a possible model and the constraints (4.6) do not "fire". So to guarantee that the actions in the

plan are always executable we have to add more rules to $P^{LP}{\downarrow}$:

$$notex :- \texttt{not } a_{0,1}(\overline{X_{0,1}}, 0), \texttt{not } \neg a_{0,1}(\overline{X_{0,1}}, 0).$$
$$notex :- \texttt{not } a_{0,2}(\overline{X_{0,2}}, 0), \texttt{not } \neg a_{0,2}(\overline{X_{0,2}}, 0).$$
$$\vdots$$
$$notex :- \texttt{not } a_{0,k_0}(\overline{X_{0,k_0}}, 0), \texttt{not } \neg a_{0,k_0}(\overline{X_{0,k_0}}, 0).$$
$$\vdots$$
$$notex :- \texttt{not } a_{n-1,k_{n-1}}(\overline{X_{n-1,k_{n-1}}}, n-1), \texttt{not } \neg a_{n-1,k_{n-1}}(\overline{X_{n-1,k_{n-1}}}, n-1).$$

2. Furthermore, in all situations whenever a constraint is violated or an inconsistency arises, the existence of a model serving as a witness for insecurity can be prevented. Thus, the translation of constraint rules has to be modified analogously: All constraints (rules with empty head) emerging from the translation above have to be rewritten to rules with only *notex* in the head: This is necessary for all rules (4.3) "`caused false...`" but not for initial state constraints (4.4), so we do not rewrite rules for the initial state.

   For example
   `:- on(B,L,T1), on(B,L1,T1), L<>L1, time(T1).`
   from the example above is rewritten to

   ```
   notex :- on(B,L,1), on(B,L1,1), L<>L1, time(1).
   notex :- on(B,L,1), on(B,L1,1), L<>L1, time(2).
   ⋮
   notex :- on(B,L,1), on(B,L1,1), L<>L1, time(n).
   :- on(B,L,0), on(B,L1,0), L<>L1.
   ```

   for all ground instantiations of the constraint, where `n` is the max. plan length.

   Fluent inconsistencies are caught by introducing rules:

   $$notex :- f(\overline{X}, T), \neg f(\overline{X}, T)$$

   for all legal fluent instantiations $f(\overline{X})$ and time steps $T > 0$.

3. When we enforce sequential planning like described above also the constraints (4.2.1) and (4.2.1) have to be rewritten to LP rules with only *notex* in the head, to exclude concurrent actions, shown here for the example given above:

   `notex :-` $\texttt{move}(X_1, X_2, T)$, $\texttt{move}(Y_1, Y_2, T)$, $X_1 \neq Y_1$.
   `notex :-` $\texttt{move}(X_1, X_2, T)$, $\texttt{move}(Y_1, Y_2, T)$, $X_1 \neq Y_1$.

Finally modify the goal constraint to:

$$: - \ goal, \texttt{not}\ notex.$$

So the default negated auxiliary predicate *notex* in the goal finally guarantees that all actions in the plan must be executable and that the computation complies with the constraints.

So someone can read the rewritten goal as follows:
"If

(i) the goal is not guaranteed to be reached executing all actions in plan p or

(ii) some actions in p cannot securely be executed or

(iii) a constraint is violated when executing the actions in p

then the plan is NOT secure."
This satisfies all conditions for plan security given on page 45.

Now taken the previous example from section 4.1.6 let us check plan $p_2 = \langle\{shoot(axel)\}\rangle$. The rewritten program for checking the plan p_2 is shown in Figure 4.3. This program has a model:

$$m = \{person(axel), time(0), actiontime(0), alive(axel, 0), shoot(axel, 0), \\ time(1), -alive(axel, 1)\}$$

which serves as a witness proving plan p_2 to be NOT secure.

On the other hand, the rewritten program for the empty plan p_1, shown in 4.2 has no model, and therefore the plan is secure, i.e. to do nothing lets *axel* survive for sure.

Limitations of the Current Approach

The current approach only works for domains, where for each non-goal state there exists at most one successor state. This is why we have to restrict ourselves to stratified domains where this condition is fulfilled trivially.

$time(0).$
$time(1).\ actiontime(0).$
$goal : -alive(axel, 1).$
$: -goal, \texttt{not}\ notex.$

$alive(axel, 0) : -.$
$alive(axel, 1) : -\texttt{not}\ \neg alive(axel, 1), alive(axel, 0), person(axel), actiontime(0).$
$alive(axel, 1) \vee \neg alive(axel, 1) : -shoot(axel, 0), person(axel), actiontime(0).$

$notex : -alive(axel, 1), \neg alive(axel, 1)$

Figure 4.2: Rewritten program for checking plan p_1.

$time(0).$
$time(1).\ actiontime(0).$
$goal : -alive(axel, 1).$
$: -goal, \texttt{not}\ notex.$

$shoot(axel, 0) \vee \neg shoot(axel, 0) : -actiontime(0), person(axel).$

$alive(axel, 0) : -person(axel).$
$alive(axel, 1) : -\texttt{not}\ \neg alive(X, 1), alive(axel, 0), person(axel), actiontime(0).$
$alive(axel, 1) \vee \neg alive(axel, 1) : -shoot(axel, 0), person(axel), actiontime(0).$

$: -\neg shoot(axel, 0).notex : -\texttt{not}\ shoot(axel, 0), \texttt{not}\ \neg shoot(axel, 0).$

$notex : -alive(axel, 1), \neg alive(axel, 1)$

Figure 4.3: Rewritten program for checking plan p_2.

If the domain would not be stratified there might be examples where the current approach to check plan security might fail due to illegal successor states. That is why planning domains have to meet with the restriction of stratification for secure planning.

As a simple example for a non-proper unstratified domain where the proposed check does not work, take the planning problem P_{noprop} consisting of the following causation rules and query:

```
initially total f.
caused x if not y after f.
caused y if not z after f.
caused z if not x after f.
executable a.
caused g after a.

g?  (1)
```

a is an action in this example and x, y, z, f and g are fluents respectively. The current translation of this program P^{LP}_{noprop} is shown in Figure 4.4 (for better readability the *time* and *actiontime* rules have been skipped here). This program has obviously only one model, which includes the goal: $\{a(0), g(1), goal, \neg f(0)\}$. Though still, plan $p = \langle\{a\}\rangle$ is not secure:
Taken the legal initial state $\{f\}$, $\{a\}$ *is* an executable action set, but there is no successor state, hence no legal transition exists, which contradicts plan security. But, if there is no successor state, the security check for plan $p = \langle\{a\}\rangle$ will falsely state that the plan is secure. The current translation of the security check tries to find a model for the program in Figure 4.5. Unfortunately the

$a(T) \vee \neg a(T), actiontime(T).\ f(0) \vee \neg f(0).$
$x(T1)\ :-\ \texttt{not}\ y(T1), f(T), actiontime(T),\ T1 = T + 1.$
$y(T1)\ :-\ \texttt{not}\ z(T1), f(T), actiontime(T),\ T1 = T + 1.$
$z(T1)\ :-\ \texttt{not}\ x(T1), f(T), actiontime(T),\ T1 = T + 1.$
$g(T1)\ :-\ a(T), time(T),\ T1 = T + 1.$

$goal : -g(1).$
$: -\texttt{not}\ goal.$

Figure 4.4: Translation of a domain P_{noprop} to DLP.

resulting program has no model due to the inconsistency in the successor state, when f is guessed initially. Therefore plan p is falsely assumed to be secure. So the current implementation can only deal with domains, where the existence of a successor state can be determined easily. The current check only looks for failed constraints or simple inconsistencies by contradictory literals, that is why we have to require stratified domains in the current approach of checking security. Obviously, domain P_{noprop} is not stratified.

Security Check for domains with `total` rules: Note that the `total` statement, if not preceded by `initially`, also does not meet the restriction to stratified domains, which is obvious when considering the corresponding pair of causation rules.

So in general the security check proposed here, also might fail for domains with `total` rules. For example the simple domain:

```
total g.
caused false if -g.

g?  (1)
```

would have no secure plan, according to the current translation. The constraint `caused false if -g`, rewritten to

$: -g(0).$
$notex : - - g(1).$

would "fire" *notex* in case $-g$ is guessed at time 1, and therefore the empty plan would not be stated as secure by the current check.

At least, even if we allow `total` statements, the check works in terms of soundness, i.e. a plan stated to be secure **is** indeed secure, nevertheless it is not complete as it is too cautious: In general, not all secure plans are found, as shown with this example. However, in Chapter 6 we will give an example, where the security check works even in presence of `total`. The problem obviously only occurs, when allowing constraints and total rules in conflict.

$a(0) \vee \neg a(0), actiontime(0).$
$f(0) \vee \neg f(0).$
$\neg a(0).$
$notex : -\texttt{not}\ a(0), \texttt{not}\ \neg a(0).$
$f(0) \vee \neg f(0).$
$x(1) \ :- \ \texttt{not}\ y(1), f(0), actiontime(0),\ 1 = 0 + 1.$
$y(1) \ :- \ \texttt{not}\ z(1), f(0), actiontime(0),\ 1 = 0 + 1.$
$z(1) \ :- \ \texttt{not}\ x(1), f(0), actiontime(0),\ 1 = 0 + 1.$
$g(1) \ :- \ a(0), time(0),\ 1 = 0 + 1.$
$notex : -f(0), \neg f(0).$
$notex : -g(0), \neg g(0).$
$notex : -x(0), \neg x(0).$
$notex : -y(0), \neg y(0).$
$notex : -z(0), \neg z(0).$
$notex : -f(1), \neg f(1).$
$notex : -g(1), \neg g(1).$
$notex : -x(1), \neg x(1).$
$notex : -y(1), \neg y(1).$
$notex : -z(1), \neg z(1).$

$goal : -g(1).$
$: -goal, \texttt{not}\ notex.$

Figure 4.5: Checking plan $p = \langle \{a\} \rangle$ for domain P_{noprop}

Security Check for domains with mutual exclusive rules: As mentioned above, pairs of positive and negative inertia of the same fluent are also forbidden, as they harm the demand for stratification. Nevertheless, pairwise inertia for a fluent f could be allowed safely, as inertia depends on the state of the fluent f in the preceding state. So for

```
inertial f.
inertial -f.
```

which corresponds to the rules:

```
caused f if not -f after f.
caused -f if not f after -f.
```

only one of both rules can fire. So it would be possible to take advantage of this mutually exclusivity of rules: A solution for this problem would be to check stratification of a domain PD for both rules: Take the remaining rules of the domain together with the positive inertia rule, denoted as domain PD', then take the remaining rules together with the negative inertia rule, denoted as domain PD''. The security check will work, if both domains PD' and PD'' are stratified. This method can be applied not only for inertia but for all pairs of rules, where the `after` parts are not satisfiable simultaneously.

4.2.3 Optimization for Sequential Planning through Binarization

In the rest of this section we examine a possible optimization of the translation which has not yet been implemented in the prototype introduced in the next chapter. Nevertheless performance tests with manual encoding of planning domains in DLV have shown that the optimized translation presented here could be a valuable extension of the current implementation (see Chapter 7).

In general for logic programs the size of the grounded program is an immanent speed factor, so if we can efficiently reduce the size of the ground instantiation we can speed up the computation of models and plans. The idea for the following optimization is due to a side note in [33]. They state that "Taking c the the number of elements (constants) in the largest type, d the maximum depth of quantifier nesting in any schema, and k to be the number of literals in the longest schema, the total length of the instantiated theory is bounded by $\mathcal{O}(kc^d)$." This upper bound also holds for an LP program, where c analogously is the number of constants in the rule, d is the maximum number of variables occurring in a rule, and k is the maximum number of literals in a rule. Under sequential planning for any action predicate the time variable T in the translation somehow is the "key" for the action predicate identifying the unique action at time T. So due to Kautz' and Selman's idea "to replace predicates that take three or more arguments by several predicates that take no more than two arguments" we can split all action predicates a with arity ≥ 2. For example let us extend the $shoot(X)$ action from above to a binary action $shoot(G, P)$

which expresses that someone shoots at person P with using G which would result in an ternary LP predicate $shoot(G, P, T)$ that can be split to two binary predicates $shoot_1(G, T)$, $shoot_2(P, T)$.

We have to adapt the translation as follows:

For each `executable` statement e (rewritten due to section 3.1.7):

$$\texttt{executable}\ a(\overline{X})\ \texttt{if}\ b_1(\overline{X_1}), \ldots, b_m(\overline{X_m}), \texttt{not}\ b_{m+1}(\overline{X_{m+1}}), \ldots, \texttt{not}\ b_n(\overline{X_n})$$

we add the following rules:

$$\begin{aligned}
&a_1(Y_1, T) \ \vee\ \neg a_1(Y_1, T) \ :- \ B_1, actiontime(T).\\
&a_2(Y_2, T) \ \vee\ \neg a_2(Y_2, T) \ :- \ B_2, a_1(Y_1, T).\\
&\vdots\\
&a_k(Y_k, T) \ \vee\ \neg a_1(Y_k, T) \ :- \ B_k, a_1(Y_1, T), a_2(Y_2, T), \ldots\ , a_{k-1}(Y_{k-1}, T).
\end{aligned}$$

where $\overline{X} = (Y_1, Y_2, \ldots, Y_k)$, i.e. arity of a is k. B_i denotes the subset of the body in rule 4.2, with all literals successively eliminated, which do not contain the variables Y_i, Y_{i-1} nor are transitively bound with another predicate which depends on Y_i. To further improve the translation, we do not have to add all $a_1(Y_1, T), a_2(Y_2, T), \ldots\ , a_{i-1}(Y_{i-1}, T)$ in the body for guessing a_i, but it is sufficient to add those which contain variables bound in B_k.

Now we have to ensure that only complete actions occur, i.e. that no action is only partly executed. So for each action a with arity k we also add the rule

$$action(T) \ :- \ a_k(Y_k, T).$$

and finally we add the constraint

$$:-\texttt{not}\ action(T), actiontime(T). \tag{4.7}$$

The auxiliary predicate $action(T)$ guarantees that a complete action is taken in each state.

Now we have to add constraints to ensure that no concurrent actions occur: For each action a with arity k we add the following constraints:

$$\begin{aligned}
&:- \ a_1(X, T), a_1(X', T), X \neq X'.\\
&:- \ a_2(X, T), a_2(X', T), X \neq X'.\\
&\vdots\\
&:- \ a_k(X, T), a_k(X', T), X \neq X'.
\end{aligned}$$

Furthermore, for each pair of action names a, a' we add the constraint:

$$:- a_1(X, T), a_1'(Y, T).$$

which forbids concurrent actions on the one side and on the other eliminates partly execution of an action. Together with constraint 4.7 this guarantees that exactly one complete action is executed at each point in time.

Note that there is a possibly important semantic difference between this optimized translation and the original one, as the optimized translation only accepts plans with exactly one action taken a time whereas the original definition of sequential plans allows transitions without any actions occurring. Anyway this can easily be remedied by introducing a "*nop*" action, which has no effects at all.

Furthermore, we now have to adapt the causation rules and initial state constraints accordingly: For each causation rule c (before rewriting it) with a non-empty `after` part [1]

$$\texttt{caused } f(\overline{X}) \texttt{ if } b_1(\overline{X_1}),\ldots,b_k(\overline{X_k}),\texttt{not } b_{k+1}(\overline{X_{k+1}}),\ldots,\texttt{not } b_l(\overline{X_l})$$
$$\texttt{after } a_1(\overline{Y_1}),\ldots,a_m(\overline{Y_m}),\texttt{not } a_{m+1}(\overline{Y_{m+1}}),\ldots,\texttt{not } a_n(\overline{Y_n})$$

we substitute each $a_i(\overline{Y_i}) \in \mathcal{L}_{act}$, $i \in \{1,2,\ldots,n\}$, where $\overline{Y_i} = (Y_1, Y_2, \ldots)$ with the corresponding (possibly default negated) unary literals $a_{i,1}(Y_1), a_{i,2}(Y_2)\ldots$ where we take only the literals where Y_j is bound in some other literal in the rule. The resulting causation is translated straightforward like above.

For instance take the action $shoot(G,P)$ from above, which has two parameters and let us assume the following $\mathcal{K}$ representation of action *shoot* and its effects:

$alive(P)$ `requires` $person(P)$.
$loaded(G)$ `requires` $gun(G)$.
$shoot(G,P)$ `requires` $gun(G)$,$person(P)$.
`executable` $shoot(G,P)$ `if` $loaded(G)$, $gun(G)$, $person(P)$.

The fluent *loaded* describes that gun G is loaded, *gun* and *person* represent type predicates defined in the background knowledge. The basic translation of the `executable` statement under sequential planning would result in:

$$shoot(G,P,T) \vee \neg shoot(G,P,T) :- actiontime(T), gun(G), person(P).$$
$$:- shoot(G,P,T), shoot(G',P',T), G \neq G'.$$
$$:- shoot(G,P,T), shoot(G',P',T), P \neq P'.$$

Applying the split technique for the binary *shoot*-action would result in:

$$shoot_1(G,T) \vee \neg shoot_1(G,T) :- loaded(G), gun(G), actiontime(T).$$
$$shoot_2(P,T) \vee \neg shoot_2(P,T) :- person(P), shoot_1(G,T).$$
$$action(T) :- shoot_2(P,T).$$
$$:- \texttt{not } action(T), actiontime(T).$$
$$:- shoot_1(X,T), shoot_1(X',T), X \neq X'.$$
$$:- shoot_2(X,T), shoot_2(X',T), X \neq X'.$$

[1] Actions may only occur in the `after` part

the effects of *shoot* [2] are described with:

```
caused -alive(P) after shoot(G,P).
caused -loaded(G) after shoot(G,P).
```

again let us first take a look at the basic translation:

$-\ alive(P,T1)\ :-\ shoot(G,P,T), person(P), actiontime(T), T1 = T+1.$
$-\ loaded(G,T1)\ :-\ shoot(G,P,T), gun(P), actiontime(T), T1 = T+1.$

whereas in the version using the splitting technique this would look as follows:

$-\ alive(P,T1)\ :-\ shoot_2(P,T), actiontime(T), T1 = T+1.$
$-\ loaded(G,T1)\ :-\ shoot_1(G,T), actiontime(T), T1 = T+1.$

What we can see here, is an improvement similar to projection in databases, a similar improvement for logic programming in general is described in [22]: Intuitively only the "relevant parts" of an action predicate are used in the translated rules, and action guesses only use binary predicates, which both reduces the size of the grounded program.

[2] For simplicity reasons we do not assume `total` nondeterminism here, like above.

Chapter 5

The Planning System DLV$^{\mathcal{K}}$

We have implemented a fully operational prototype supporting the $\mathcal{K}$ language as a frontend on top of the DLV system [18]. This frontend is invoked by the command-line option `-FP` of DLV. It reads $\mathcal{K}$ files, that is, files as described in the following subsection, whose names carry the extension `.plan`, and optionally also background knowledge in the form of stratified datalog programs and transforms these into the core language of DLV using the translation from Chapter 4. The frontend then invokes the DLV kernel and translates possible solutions back into output appropriate for the planning user. Additionally in the current implementation it is possible to accept input from the $\mathcal{C}$ action language.

5.1 DLV Core Language and Syntax

The core language of DLV is disjunctive datalog (function-free logic programming) under the Answer Set semantics [26] with integrity constraints , strong negation, and queries, as described in Section 2.1. Integer arithmetics and various builtin predicates are also included in the core language. Rules have the form:

$$a_1 \texttt{ v } \ldots \texttt{v } a_m \texttt{ :- } b_1, \ldots, b_k, \texttt{not } b_{k+1}, \ldots, \texttt{not } b_n$$

where $m, n \geq 0$. $a_1, \ldots a_m, b_1, \ldots, b_n$ are literals $a(t_1, \ldots, t_f)$ or a strongly negated atom $\texttt{-}a(t_1, \ldots, t_f)$.

As usual, constants are starting with a lower case letter, while variables are starting with an upper case letter.

The planning frontend accepts such DLV programs as input for the background knowledge, and planning input of the form presented in the subsequent sections will be translated to DLV programs.

5.2 DLV$^{\mathcal{K}}$ Frontend

5.2.1 Synopsis and Command Line Options:

The planning frontend can be invoked from the command line using the following syntax:

```
$ dl [input-files] [options]
```

input-files Allowed input files are

- normal DLV programs consisting of DLP rules and constraints (background knowledge).
- planning input files described below, with file name extension `.plan` (All files with this extension are automatically treated as planning input.)

options To invoke the planning frontend at least one of the following options has to be used:

-FP planning with $\mathcal{K}$ input in interactive mode (for each optimistic plan found, the user will be asked, whether plan security should be checked, and whether DLV should continue to search for other plans.)

-FPopt find optimistic plans in batch mode. Optimistic plans will be printed without user interaction.

-FPsec find only secure plans in batch mode. Only secure plans will be printed without user interaction. This works only for stratified domains, the current method for checking whether domains are stratified is described in detail in Section 5.3.3

-FPc For historical reasons using this option DLV also accept a subset of Lifschitz' $\mathcal{C}$ Action language as input (see Section 2.2 for Details), i.e. what is described as "definite" domains there. The translation of $\mathcal{C}$ input, is in accordance with [35].[1]

In addition to these options one can specify the maximum plan length from the command-line using option **-planlength=**i where i is a positive integer. A full description of all DLV command-line options can be found in the DLV online manual [23].

5.2.2 DLV$^{\mathcal{K}}$ Programs

A $\mathcal{K}$ planning domain, as implemented in DLV$^{\mathcal{K}}$, consists of various (optional) sections that start with a keyword followed by a colon. The overall structure of a $\mathcal{K}$ program is shown in Figure 5.1. *<fluent declarations>* and *<action declarations>* are sequences of declarations as defined in Section 3.1.3, and

[1] For $\mathcal{C}$ input special $\mathcal{K}$ language features like `total`, `executable` or default negation can not be used. Also the keyword `securePlan` is not allowed.

```
fluents:    <fluent declarations>
actions:    <action declarations>
always:     <rules>
initially:  <init rules>
            [noConcurrency.]    [securePlan.]
goal:       <query>?(i)
```

Figure 5.1: Structure of DLV$^{\mathcal{K}}$ programs

<rules> is a sequence of causation rules (including the enhanced statements using keywords `total, default, nonexecutable`) and executability conditions which apply to any state. *<init rules>* is a sequence of static rules which apply to the initial state only, that means initial state constraints (including the enhanced statements using `total, default`).

The declaration of the plan length in parentheses at the end of the goal query is optional. As mentioned above plan length can also be set from the command-line using option **-planlength**=i. In case the plan length is declared in the input file this overrides the command-line option. Plan length defaults to 0 when not declared.

Comments in DLV$^{\mathcal{K}}$ programs are preceded by '%' and might be placed anywhere in the input file.

By default, DLV$^{\mathcal{K}}$ will look for plans allowing concurrent actions (that is, plans that may contain transitions $\langle s, A, s' \rangle$ with $|A| > 1$). By specifying

```
noConcurrency.
```

the user can ask for sequential plans only. In the presence of the keyword

```
securePlan.
```

or the command-line option `-FPsec`, DLV$^{\mathcal{K}}$ will only compute secure plans, as opposed to the default situation where all (optimistic) plans are computed and the user interactively decides whether to check their safety.

5.3 Implementation

5.3.1 Programming Environment

DLV is fully implemented in C++ and the planning frontend has been integrated into the core program. The scanner and parser for the frontend have been implemented using the scanner generator `flex`, a free version of `lex`, which ships with many UNIX derivates, and `yacc`, a free version of the parser generator `bison`.[2]

[2] `flex` and `yacc` and the can be downloaded at `http://www.gnu.org`

5.3.2 DLV Kernel

The internal language of the DLV system is function-free DLP with true negation [26] and strong and weak integrity constraints [21] (extended DLP). The kernel is an efficient engine for computing answer sets of a program, described in [49],[18].

The system architecture is shown in Figure 5.2. The core system accepting extended DLP as input consists of the *Query Processor* (QP), which controls system execution and together with the integrated frontends performs preprocessing of the input, and postprocessing of the generated models. First the input program is passed to the *Rules and Graph Handler* (RGH) which splits the input into subprograms. These subprograms are dispatched to the *Intelligent Grounding Module*, which efficiently generates an optimized *ground program*, which has the same stable models, but is in general much smaller than the (traditionally) grounded input program. QP then again invokes RGH, which generates two partitionings into components of the grounded program. They are used by the *Model Generator* (MG) and the *Model Checker* (MC), and enable a modular efficient program evaluation. MG then produces "candidates" for stable models which are checked for stability by the MC.

5.3.3 Integration of the Planning Frontend

So now, let us see how the planning frontend integrates into this structure: Files with suffix "`.plan`" are first processed by the *Plan Translator* (PT), which consists of the scanner and parser. PT parses the input and produces an EDLP program, according to the translation described in Chapter 4.

Simple Stratification Check Parallel with parsing the input, a dependency graph of the fluents (FDG) is produced: PT produces a node for each positive and each negative fluent name occurring in the input. For each causation rule

`caused` $G(\overline{X})$ `if` $F_1(\overline{X_1})$, ... $F_m(\overline{X_m})$,
`not` $F_{m+1}(\overline{X_{m+1}})$, ... `not` $F_n(\overline{X_n})$ `after` ...

for all F_i with $i \leq m$ a positive directed arc (F_i, G) is added to FDG, for all F_j with $m < j \leq n$ a negative directed arc is added. If this graph contains no cycles with a negative arc, the program is stratified for sure.

The frontend allows secure planning **only** if FDG does not contain cycles with negative arcs. However, as this FDG check processes only fluent names and not the ground program, which would be too expensive, proper, even stratified input could falsely be rejected for secure planning, as we have seen in Section 6.6.

A known bug in the current implementation is that stratification is not checked correctly for the `total` statement. Instead of treating a total statement like its corresponding two causation rules, the current implementation handles total statements as follows: For each `total` statement

`total` $G(\overline{X})$ `if` $F_1(\overline{X_1})$, ... $F_m(\overline{X_m})$,
`not` $F_{m+1}(\overline{X_{m+1}})$, ... `not` $F_n(\overline{X_n})$ `after` ...

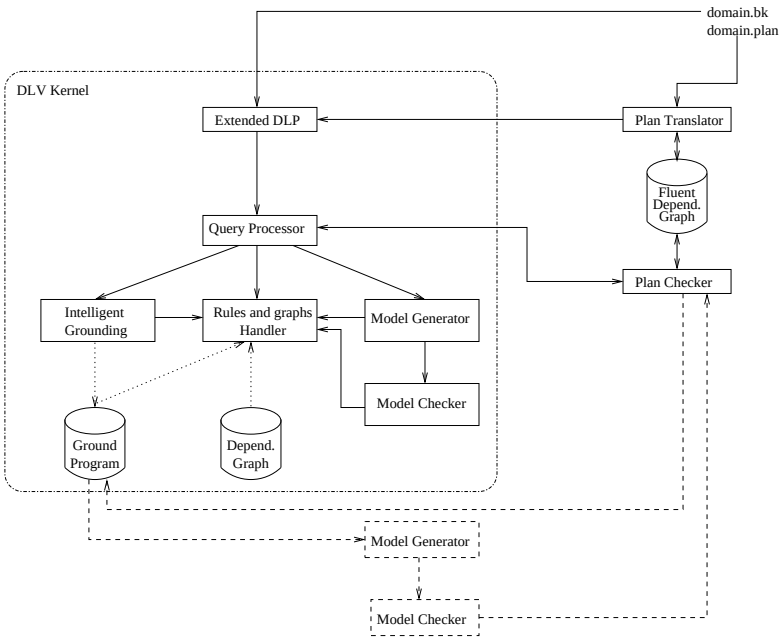


Figure 5.2: System Architecture of DLV$^{\mathcal{K}}$

for all F_i with $i \leq m$ two positive directed arcs (F_i, G),$(F_i, -G)$ are added to FDG, for all F_j with $m < j \leq n$ two negative directed arcs are added. This current approach is a tradeoff that allows to use `total` in the always section of a program, which would be forbidden in general when taking the corresponding pair of causation rules

`caused` $G(\overline{X})$ `if not` $-G(\overline{X})\ldots$
`caused` $-G(\overline{X})$ `if not` $G(\overline{X})\ldots$

instead, as those rules always build a cycle in the FDG. As mentioned in Section 4.2.2 this could cause that not necessarily all secure plans are found when using totality. However totality is escpecially important if nondeterministic action effects should be encoded. We will show an example in Section 6.7.5, where we use `total`, which works fine.

The translated program, together with the background knowledge defined in an extra file is fed into QP, which optimizes the program, produces the ground program, and generates models, like explained above.

Each generated plan is dispatched to the *Plan Checker* (PC) by QP. For optimistic planning we are ready. PC simply orders the model by the time stamps and displays the fluents and actions. Some detailed examples and a sample run of DLV$^{\mathcal{K}}$ will be presented in Chapter 6.

Under secure planning (which is activated either via the command-line option `-FPsec`, using keyword `securePlan` in the input, or in interactive mode, by manually enforcing security check), the plan checker makes a copy of the ground program which is then rewritten as described in section 4.2.2 with respect to the plan generated by QP. Afterwards PC calls another instance of MG/MC to check the rweritten program. If this rewritten program has no model, the plan is stated to be secure.

After checking plan security, the next plan delivered by QP is processed, and so forth.

5.4 Known Limitations and Bugs

As mentioned above, we cannot (yet) deal with general $\mathcal{K}$ programs, when considering secure planning, but have to restrict input to stratified programs. For a real stratification check, we would have to check the rules translation and grounding. After grounding, however, building the dependency graph would be more expensive. The current approach is a tradeoff, in accordance with the current implementation of the security check.

After all, even stratified programs are a rather small subclass of all possible proper programs. It would probably be preferable to implement an "even cycles" approach which is expensive but more general, in principle very similar to the results in [48]: According to the results there, programs with only an even number of negative arcs in each cycle of the (ground version of the) fluent dependency graph constructed while parsing are also guaranteed to be proper. This is has not yet been examined in detail so far and should be part of further investigations, as efficient graph algorithms must be implemented to check for "odd cycles".

At least it would be nice to exploit mutual exclusiveness of positive and negative inertia rules for example, a pair of rules:

```
inertial a ( = caused a if not -a after a)
inertial -a ( = caused -a if not a after -a)
```

would be rejected by the system, though they never can fire both in a state. Unfortunately in general this mutual exclusiveness of rules can only be checked on the ground level, which is beyond the current implementation, and capabilities of the current approach, which only checks on the non-ground input.

As outlined above and with the example in Section 4.2.2, the current implementation has some problems with the general use of the `total` statement, possibly not recognizing all secure plans, as totality is not treated strictly by the current implementation of the stratification check. Nevertheless, as we will see in the next chapter, many examples of nondeterministic domains using `total` can be encoded and will be solved correctly. It might be rewarding to look for stronger syntactical checks which really only allow domains, where it is guaranteed that

all secure plans will be found. On the other hand, this check should not be as strict as pure stratification.

As already pointed out in the last chapter, because of the translation possibly several models represent the same plan in nondeterministic planning domains, so plans are eventually computed more than once as each model stands for a distinct combination of states and actions (i.e. transition sequence), not only the sequence of actions itself.

Another problem with a similar cause arises, when combining DLV command-line option `-n=`i with secure planning. This option tells DLV to compute at most i answer sets for an input program. In combination with optimistic planning this is no problem, as each model corresponds to an optimistic plan, so if we want to find only the first optimistic plan, we can use option `-n=1`. However, this option tells QP to terminate after i models are found, so in general it does not work with secure planning enabled (`-FPsec`). For instance taking `n=1`, the first model/plan found is not necessarily secure, but subsequent plans found could be. So the meaning of this option is not really intuitively clear, as the user might expect that DLV searches for exactly one secure plan. Two alternative fixes are possible, either adding a new commandline option, which refers only to the number of computed plans, or adapting the existing option `-n`, which might be confusing, as it would not strictly refer to the number of computed models any longer, so the first approach seems preferable.

Furthermore, using our current prototype we cannot compute minimal length plans, but only plans with a fixed length. A fast fix to remedy this drawback, is using a shell script calling DLV in a loop increasing the argument to command-line option `-planlength` until a plan is found. Unfortunately this is rather expensive and many things are computed repeatedly, so it would be nice to integrate this feature into the DLV$^{\mathcal{K}}$ frontend.

Chapter 6

Knowledge Representation in $\mathcal{K}$

6.1 A Simple Blocks World Instance

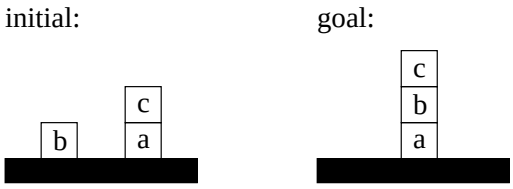


Figure 6.1: A blocks world example.

Now we are ready to present a sample run of the DLV$^{\mathcal{K}}$ frontend. We start with a short blocks world example. Referring to Figure 6.1, we want to turn the initial configuration of blocks into the goal state[1] in three steps, where only one move is allowed in each step (i.e., concurrent moves are not permitted).

First of all, the static background knowledge Π consists of the rules and actions in file "blocks.bk" (see Figure 6.2). This program describes the relevant objects in our planning domain.

The action description for the blocks world needs one action `move` and two fluents `on`, and `occupied`. We first assume that the knowledge on the initial state is complete (we know the location of all blocks) and correctly specified. We will then show how to deal with incorrect or incomplete initial state specifications. The domain is specified in file "blocks.plan" (see Figure 6.3).

Intuitively, the `executable` statement for action *move* says that a block `B` can be moved on location `L` $\neq$ `B` if both `B` and `L` are clear (note that the table

[1] The blocks world example given in this section is an implementation of the well known Sussman anomaly [55] which is similar to the implementation in [19].

```
block(a).  block(b).  block(c).
location(table).
location(B) :- block(B).
```

Figure 6.2: blocks.bk

```
fluents:   on(B,L) requires block(B), location(L).
           occupied(B) requires location(B).
actions:   move(B,L) requires block(B), location(L).
always:    executable move(B,L) if not occupied(B), not occupied(L),
                                   B <> L.
           inertial on(B,L).
           caused occupied(B) if on(B1,B), block(B).
           caused on(B,L) after move(B,L).
           caused -on(B,L1) after move(B,L), on(B,L1), L <> L1.
initially: on(a,table).  on(b,table).  on(c,a).
           noConcurrency.
goal:      on(c,b),on(b,a),on(a,table)?  (3)
```

Figure 6.3: Basic version of blocks.plan

is always clear, as it is not a block). The causation rules for `on` and `-on` specify the effect of a move. It is worthwhile noting that the totality of these fluents is not enforced (like in satisfiability planning). Both `on(x,y)` and `-on(x,y)` may happen to be not true at a given instant of time.

Actually, the rule for `-on` could be replaced by "`caused -on(B,L1) if on(B,L), L <> L1.`" stating: wherever a block is, it is not anywhere else. This rule would give us a sharper description of the state making fluent `on` total at every instant of time. Nevertheless, the extra knowledge derived for `-on` from this rule is useless for our goal, as `-on` does not appear in the body of any rule, and `-on(x,y)` is used only to override the inertial property `on(x,y)` after moving `x` from `y`. Thus, we refrain from using the more general rule which would cause a computational overhead (as more inferences are to be done during the computation) without providing relevant benefits.

The execution of this program on DLV$^{\mathcal{K}}$ computes the following:

```
PLAN: move(c,table,0), move(b,a,1), move(c,b,2)
```

Here, the additional argument in a `move` atom represents the instant of time when the action is executed, just like described in the previous chapter. Thus, the above plan requires to first move `c` on the table, then to move `b` on top of `a`, and, finally, to move `c` on `b`. It is easy to see that this sequence of actions leads to the desired goal.

To demonstrate how DLV$^{\mathcal{K}}$ is called from the command-line and how computed

plans are presented in detail, the output produced by a sample run on the files "blocks.bk" and the planning problem file "blocks.plan" is shown in Figure 6.4.

In this first simple example only one plan is found, which is obviously secure. For each plan found DLV$^{\mathcal{K}}$ displays all known fluents in each `STATE` and `ACTIONS` occurring in that state. Finally the full plan is shown, which can be tested for security before displaying the next plan.

6.2 Checking Correctness and Completeness of the Initial State

In the previous example, the knowledge on the block locations in the initial state is complete and correctly specified with respect to domain laws. To ensure that an arbitrary given (partial) knowledge state is not flawed, we should check it properly. We should here verify that every block: (i) is on top of a unique location, (ii) does not have more than one block on top of it, and (iii) is supported by the table (i.e., it is either on the table or on a stack of blocks which is on the table) [37]. To this end, we add the declaration:

```
supported(B) requires block(B).
```

And we add the following rules in the `initially` section:

```
caused false if on(B,L), on(B,L1), L<>L1.
caused false if on(B1,B), on(B2,B), block(B), B1<>B2.
caused supported(B) if on(B,table).
caused supported(B) if on(B,B1), supported(B1).
caused false if not supported(B).
```

The resulting DLV$^{\mathcal{K}}$ program does not compute any plan if the initial state is either incomplete (in the sense that not all block locations are known) or incorrectly specified. Note that, under `noConcurrency`, the action `move` preserves the properties (i),(ii), (iii) above; thus, we do not need to check these properties in all states, if concurrent actions are forbidden.

```
$ dl blocks.plan blocks.dl -FP

STATE 0:  occupied(a,0), on(a,table,0), on(b,table,0), on(c,a,0)
ACTIONS: move(c,table,0)
STATE 1:  on(a,table,1), on(b,table,1), on(c,table,1), -on(c,a,1)
ACTIONS: move(b,a,1)
STATE 2:  occupied(a,2), on(a,table,2), on(b,a,2), on(c,table,2),
-on(b,table,2)
ACTIONS: move(c,b,2)
STATE 3:  on(a,table,3), on(b,a,3), on(c,b,3), -on(c,table,3),
occupied(a,3), occupied(b,3)
PLAN: move(c,table,0), move(b,a,1), move(c,b,2)

Check whether that plan is secure (y/n)?  y
The plan is secure.

Search for other plans (y/n)?  y

$
```

Figure 6.4: Sample run of DLV$^{\mathcal{K}}$

6.3 Reasoning under Incomplete Knowledge

Suppose now that there is a further block `d` in Figure 6.1. The exact location if `d` is unknown, but we know that it is not on top of `c`.

We look for a plan that works on every possible initial state (i.e., no matter if `on(d,b)` or `on(d,table)` holds), and reaches the goal `on(a,c), on(c,d), on(d,b), on(b,table)` in four steps. We modify the background knowledge Π by adding the fact `block(d).` in "blocks.bk" and change the planning domain file:

1. change the goal q to

   ```
   on(a,c),on(c,d),on(d,b),on(b,table)?  (4)
   ```

2. add `-on(d,c)` and `total on(X,Y)` in the `initially` section

3. add the command `securePlan`.

The execution of this program on DLV$^{\mathcal{K}}$ computes the following two secure plans:

```
PLAN: move(d,table,0), move(d,b,1), move(c,d,2), move(a,c,3)
PLAN: move(d,c,0), move(d,b,1), move(c,d,2), move(a,c,3)
```

The plans are clearly valid on all possible legal initial states. Since the effects of all actions are determined, these plans are also secure.[2]

[2] When running this example each plan is displayed twice, which represents the application

Note that an optimistic 2-step plan exists: `move(c,d,0), move(a,c,1)`, as `c` could initially be on `b`. However, this plan is not secure.

6.4 Compact Representation Using Conditional Totalization

The following is an extension to the results already published in [15]. Using conditional totalization (non empty `if` part in the `total` statement), leads to an even more compact representation of the blocks world example.

Instead of adding `-on(d,c)` and `total on(X,Y)` we can add the single statement `total on(d,L) if L <> c` in the `initially` section. The final version of our blocks world planning problem in DLV$^{\mathcal{K}}$ is shown in Figure 6.5.

6.5 Translation of the Example

Finally to take a closer look at the resulting logic program of our final version of the blocks world domain, the translated rules in DLV can be examined using the command-line options `--print-edb`, `--print-idb` and `--print-constraints` to watch the generated rules and constraints of the resulting program, according to the basic translation in Chapter 4.1:

$ dl blocks.plan blocks.dl -FP –print-edb –print-idb –print-constraints

As the output of this command is not really compact and already includes some internal optimizations of DLV, a more readable version of the translated program is shown in Figure 6.6 (lines marked with '%' are comments). Each model of this program corresponds to an optimistic plan of the blocks world problem.

In order to check plan security, the DLV$^{\mathcal{K}}$ frontend internally implements the translation to DLV pointed out in Section 4.2.2. As this translation refers to the grounded program, due to space restrictions I will do without a detailed description of this rather long example here, one can refer to the two short examples in the previous chapter.

6.6 How DLV$^{\mathcal{K}}$ Deals with Unstratified Domains

As pointed out before, the secure check (see Section 4.2.2) does only work correctly for domains which are stratified. The current version of DLV$^{\mathcal{K}}$ cannot check properness in general but implements a more restrictive check, demanding stratification of the predicate symbols (on the non ground level) of a domain, details can be found in Chapter 5.

of the plan on each of the the two different legal initial states. The reason for this is that each plan found by DLV$^{\mathcal{K}}$ stands for a model of the translated program P^{LP}, and of course two different initial states represent different models

```
fluents:  on(B,L) requires block(B), location(L).
          occupied(B) requires location(B).
          supported(B) requires block(B).
actions:  move(B,L) requires block(B), location(L).
always:   executable move(B,L) if not occupied(B), not occupied(L),
                                   B <> L.
          inertial on(B,L).
          caused occupied(B) if on(B1,B), block(B).
          caused on(B,L) after move(B,L).
          caused -on(B,L1) after move(B,L), on(B,L1), L <> L1.
initially:caused false if on(B,L), on(B,L1), L<>L1.
          caused false if on(B1,B), on(B2,B), block(B), B1<>B2.
          caused supported(B) if on(B,table).
          caused supported(B) if on(B,B1), supported(B1).
          caused false if not supported(B).
          total on(d,B) if B <> c.
          on(a,table).
          on(b,table).
          on(c,a).
          noConcurrency.
          securePlan.
goal:     on(c,b),on(b,a),on(a,table)?  (4)
```

Figure 6.5: The final version of blocks.plan

```
% background knowledge:
block(a).
block(b).
block(c).
block(d).
location(B) :- block(B).
location(table).

% time predicates
actiontime(0).  actiontime(1).  actiontime(2).  actiontime(3).
nexttime(0,1).  nexttime(1,2).  nexttime(2,3).  nexttime(3,4).
time(0).  time(1).  time(2).  time(3).  time(4).

% rules and constraints:
-move(B,L,T) v move(B,L,T) :- !=(B,L), actiontime(T), block(B),
location(L), location(B), not occupied(B,T),not occupied(L,T).
on(B,L,T1) :- on(B,L,T), nexttime(T,T1), block(B), location(L),
not -on(B,L,T1).
occupied(B,T) :- on(B1,B,T), block(B), time(T), location(B).
on(B,L,T1) :- move(B,L,T), nexttime(T,T1), block(B), location(L).
-on(B,L,T1) :- move(B,L1,T), on(B,L,T), !=(L1,L), nexttime(T,T1),
block(B), location(L).
supported(B,0) :- on(B,table,0), block(B).
supported(B,0) :- on(B,B1,0), supported(B1,0), block(B).
-on(d,B,0) v on(d,B,0) :- block(d), location(B), !=(B,c).
-on(d,c,0) :- block(d), location(c).
on(a,table,0) :- block(a), location(table).
on(b,table,0) :- block(b), location(table).
on(c,a,0) :- block(c), location(a).
:- on(B,L,0), on(B,L1,0), !=(L,L1).
:- on(B1,B,0), on(B2,B,0), block(B), !=(B1,B2).
:- block(B), not supported(B,0).

% goal query
goal :- on(a,c,4), on(c,d,4), on(d,b,4), on(b,table,4).
:- not goal.

% noConcurrency
:- move(B1,L1,T), move(B2,L2,T), !=(B1,B2).
:- move(B1,L1,T), move(B2,L2,T), !=(L1,L2).
```

Figure 6.6: Translation of the final version of the blocks world problem.

Recalling P_{noprop} from the last chapter, (see Figure 4.4) the DLV$^{\mathcal{K}}$ representation of this problem is:

```
fluents:  f.  g.  x.  y.  z.
always:   caused x if not y after f.
          caused y if not z after f.
          caused z if not x after f.
          executable a.
          caused g after a.
initially:total f.
goal:     g?  (1)
```

DLV$^{\mathcal{K}}$ recognizes that the domain is not stratified, and computes only optimistic plans:

```
$ dl noprop.plan -FP

Cannot check whether plans are secure because the domain is
probably not proper.
STATE 0:  -f(0)
ACTIONS: a(0)
STATE 1:  g(1)
PLAN: a(0)
```

Unfortunately, the restrictive policy also rules out actually stratified programs, for instance, the rule:

```
caused holds_spoon(righthand) if not holds_spoon(lefthand).
```

would be forbidden, as the parameters `lefthand, righthand` are not considered by the stratification check.

On the other hand the current check does not recognize violations of stratification due to the `total` statement in general. The example from Section 4.2.2, would be accepted as input:

```
fluents:  g.
always:   total g.
          caused false if -g.
goal:     g?  (1)
```

Anyway, DLV$^{\mathcal{K}}$ does not find a secure plan on this example.

6.7 Further Examples of Problem Solving in $\mathcal{K}$

This section contains some more encodings of planning problems from the literature, which should further illustrate the practical use of language $\mathcal{K}$ and how well-known planning problems can be described with our language. In particular, Section 6.7.3 discusses concurrent actions, an example in Section 6.7.5 shows the capabilities of DLV$^{\mathcal{K}}$ dealing with nondeterministic action effects.

6.7.1 The Yale Shooting Problem

Another example for dealing with incomplete initial knowledge is an encoding of the famous Yale Shooting Problem (see [32]), which has in some variations already appeared in the previous chapters. We assume here that the agent has a gun and does not know whether it is initially loaded. This can be modeled as follows:

```
fluents:    alive.  loaded.
actions:    load.  shoot.
always:     executable shoot if loaded.
            executable load if not loaded.
            caused -alive after shoot.
            caused -loaded after shoot.
            caused loaded after load.
initially:  total loaded.
            alive.
goal:       -alive?(1)
```

Obviously, the `total` statement leads to two possible legal initial states: {`loaded, alive`} and {`-loaded, alive`}. The first one enables an optimistic plan executing `shoot`; however, because of the second one, this plan is not secure. As desired, DLV$^{\mathcal{K}}$ produces the following output:

```
PLAN: shoot(0)
Check whether that plan is secure (y/n)?  y
The plan is NOT secure.
```

6.7.2 The Monkey and Banana Problem

This example is a variation of the Monkey and Banana problem as described in the *CCALC* manual (<URL:http://www.cs.utexas.edu/users/mccain/cc/>). It shows that in $\mathcal{K}$ applicability of actions can be formulated very intuitively through the `executable` statement. The encoding in *CCALC* uses many `nonexecutable` statements instead.

In the background knowledge we have three objects: the monkey, the banana and a box:

```
object(box).
object(monkey).
object(banana).
```

The planning world has locations numbered with integers. In the beginning, the monkey is at location 1, the box is at location 2, and the banana is hanging from the ceiling over location 3. The monkey shall get the banana by moving the box towards it, climbing the box, and then grasping the banana hanging from the ceiling. We solve this problem using the following DLV$^{\mathcal{K}}$ program:

```
fluents:    at(0,L) requires object(0),#int(L).
            onBox.
            hasBanana.
actions:    walk(L) requires #int(L).
            pushBox(L) requires #int(L).
            climbBox.
            graspBanana.
always:     caused at(monkey,L) after walk(L).
            caused -at(monkey,L) after walk(L1), at(monkey,L), L<>L1.
            executable walk(L) if not onBox.

            caused at(monkey,L) after pushBox(L).
            caused at(box,L) after pushBox(L).
            caused -at(monkey,L) after pushBox(L1), at(monkey,L), L<>L1.
            caused -at(box,L) after pushBox(L1), at(box,L), L<>L1.
            executable pushBox(L) if at(monkey,L1), at(box,L1), not onBox.

            caused onBox after climbBox.
            executable climbBox if not onBox, at(monkey,L), at(box,L).
            caused hasBanana after graspBanana.
            executable graspBanana if onBox, at(monkey,L), at(banana,L).

            inertial at(0,L).
            inertial onBox.
            inertial hasBanana.
initially:  at(monkey,1).
            at(box,2).
            at(banana,3).
noConcurrency.
goal:       hasBanana ?  (4)
```

For this planning problem, DLV$^{\mathcal{K}}$ finds the following plan, which is secure:

```
PLAN: walk(2,0), pushBox(3,1), climbBox(2), graspBanana(3)
```

Let us now show how to check correctness of the initial state. Similar to the Blocks World example in Section 6.2, we introduce a new fluent:

```
objectIsSomewhere(0) requires object(0).
```

Furthermore, we add the following constraints and rules in the initial state:

```
initially: caused false if at(O,L), at(O,L1), L<>L1.
           caused false if onBox, at(monkey,L), not atBox(L).
           caused objectIsSomewhere(O) if at(O,L).
           caused false if not objectIsSomewhere(O).
```

These constraints guarantee a correct initial state. Similar to the Blocks World domain, we could now for example allow the user to specify uncertain initial knowledge about the position of the monkey by replacing

```
at(monkey,1).
```

with:

```
total at(monkey,X).
```

in the `initially` section.

6.7.3 The Rocket Transport Problem

This example is a variation of a planning problem for rockets introduced by Veloso [57]. There are two one-way rockets, which can transport cargo objects from one place to another. The objects have to be loaded on the rocket and unloaded at the destination. This example shows the capability of $\mathcal{K}$ to deal with concurrent actions, as the two rockets can be loaded, can move, and can be unloaded in parallel.

The background knowledge consists of three places, the two rockets and the objects to transport:

```
rocket(sojus).  rocket(apollo).
cargo(food).  cargo(tools).  cargo(car).
place(earth).  place(mir).  place(moon).
```

The action description for the rocket planning domain comprises three actions `move(R,L)`, `load(C,R)` and `unload(C,R)`. The fluents are `atR(R,L)` (where the rocket currently is), `atC(C,L)` (where the cargo object currently is), `in(C,R)` (describing that an object is inside a rocket) and `hasFuel(R)` (the rocket has fuel and can move). Now let us solve the problem of transporting the car to the moon and food and tools to Mir, given that all objects are initially on the earth and both rockets have fuel. We write the following DLV$^{\mathcal{K}}$ program:

```
fluents:    atR(R,P) requires rocket(R), place(P).
            atC(C,P) requires cargo(C), place(P).
            in(C,R) requires rocket(R), cargo(C).
            hasFuel(R) requires rocket(R).
actions:    move(R,P) requires rocket(R), place(P).
            load(C,R) requires rocket(R), cargo(C).
```

```
            unload(C,R) requires rocket(R), cargo(C).
always:     caused atR(R,P) after move(R,P).
            caused -atR(R,P) after move(R,P1), atR(R,P).
            caused -hasFuel(R) after move(R,P).
            executable move(R,P) if hasFuel(R), not atR(R,P).

            caused in(C,R) after load(C,R).
            caused -atC(C,P) after load(C,R), atC(C,P).
            executable load(C,R) if atC(C,P), atR(R,P).

            caused atC(C,P) after unload(C,R), atR(R,P).
            caused -in(C,R) after unload(C,R).
            executable unload(C,R) if in(C,R).

            nonexecutable move(R,P) if load(C,R).
            nonexecutable move(R,P) if unload(C,R).
            nonexecutable move(R,P) if move(R,P1), P<>P1.
            nonexecutable load(C,R) if load(C,R1), R<>R1.

            inertial atC(C,L).
            inertial atR(R,L).
            inertial in(C,R).
            inertial hasFuel(R).
initially:  atR(R,earth).
            atC(C,earth).
            hasFuel(R).

securePlan.

goal:       atC(car,moon), atC(food,mir), atC(tools,mir)?(3)
```

The `nonexecutable` statements exclude simultaneous actions as follows:

- loading/unloading a rocket and moving it;
- moving a rocket to two different places;
- loading an object on two different rockets.

For the given goal, DLV$^{\mathcal{K}}$ finds two secure plans, where in the first one rocket `sojus` flies to the moon and `apollo` flies to Mir, and in the second one the roles are interchanged:

```
PLAN: load(food,sojus,0), load(tools,sojus,0), load(car,apollo,0),
      move(sojus,mir,1), move(apollo,moon,1),
      unload(food,sojus,2), unload(tools,sojus,2),
      unload(car,apollo,2)
PLAN: load(car,sojus,0), load(food,apollo,0), load(tools,apollo,0),
      move(sojus,moon,1), move(apollo,mir,1),
      unload(car,sojus,2), unload(food,apollo,2),
      unload(tools,apollo,2)
```

6.7.4 The Towers of Hanoi

The next example is Towers of Hanoi, a game with three pegs and a variable number of disks, which have to be moved from on peg to another. All the disks have different sizes and a disk may be moved only to an empty peg or onto a larger disk. The variable number of disks can be encoded elegantly, like shown in the following program defining the background knowledge:

```
% 3 pegs, n disks
peg(p1).
peg(p2).
peg(p3).

location(X) :- peg(X).
location(X) :- #int(X).

smaller(D,D1) :- #int(D), #int(D1), D < D1.
smaller(D,P) :- #int(D), peg(P).
```

This program allows us to tell DLV how many disks should be stapled by using command-line option `-N=`n: `#int` is a builtin predicate in DLV, which is true for all positive integers form 0 to n. So the meaning of the program above is that the disks are numbered from 0 to n, for example using "-N=2" means we have three disks: 0,1 and 2.

The action description is very similar to the blocks world example with one action `move` and fluents and fluents `on` and `occupied`. The fluent `unfinished` indicates, whether we have reached the goal:

```
fluents:  on(D,L) requires #int(D), location(L).
          occupied(L) requires location(L).
          unfinished.

actions:  move(D,L) requires #int(D), location(L).

always:   executable move(D,L) if not occupied(D), not occupied(L),
                                  smaller(D,L).
          inertial on(D,L).
          caused occupied(L) if on(D,L).
          caused on(D,L) after move(D,L).
          caused -on(D,L1) after move(D,L), on(D,L1).
          caused unfinished if not on(#maxint,p2).
          caused unfinished if not on(D,D1), #succ(D,D1),
                               D < #maxint.

initially:caused on(D,D1) if #int(D1),#succ(D,D1).
          on(#maxint,p1).

noConcurrency.

goal:     not unfinished?
```

`#maxint` is a builtin constant in DLV, automatically set to the number given with command-line option -N=n, the predicate `#succ` represents the successor relation of integers.

This example shows how the command-line options -planlength=i and -N=n can be used to scale up planning domains. For instance, if we have three disks we need at least seven steps to reach the goal. So we have to invoke DLV using the following command:

```
$ dl hanoi.plan hanoi.bk -FP -N=2 -planlength=7
```

DLV$^{\mathcal{K}}$ finds the following (secure) plan, which is the expected solution:

```
PLAN: movedisk(0,p2,0), movedisk(1,p3,1), movedisk(0,1,2),
      movedisk(2,p2,3), movedisk(0,p1,4), movedisk(1,2,5),
      movedisk(0,1,6)
```

6.7.5 The Bomb in the Toilet

The last problem instance we examine here shows that DLV$^{\mathcal{K}}$ can also deal with nondeterministic action-effects under certain restrictions, not only incomplete initial knowledge, and thus can also solve the "bomb in the toilet" problem [44], a problem often studied in conformant planning (see [53],[10],[11],[5],[24]). The problem has several formulations, which can be described as follows: We have been alarmed that there is a bomb (at most one) in a lavatory. There are p suspicious packages which could contain the bomb. There are t toilet bowls, which can be clogged or not. Possible actions are: Flushing a toilet, which is always executable and has the effect that the toilet is not clogged. Dunking a package into a toilet requires the toilet to be unclogged and has the effect of disarming the bomb if the package contains one. Dunking might also clog the toilet (but it might as well not clog it).

In some formulations (but not all) the toilets are known to be unclogged in the initial state, in others their clogged state is unknown. The goal is always to disarm the bomb. We will now take a closer look at different formulations of this problem, the naming convention of the domains is due to [11]:

BT(p) - Bomb in toilet with p packages The first domain we will observe, is the classical bomb in the toilet, where there is only uncertainty in the initial state, i.e. it is not known which one of p packages contains the bomb, and there is no notion of clogging:

```
fluents:  armed(P) requires package(P).
          dunked(P) requires package(P).
          unsafe.
actions:  dunk(P,T) requires package(P), toilet(T).
always:   inertial armed(P).
          caused dunked(P) after dunked(P).
```

```
          caused -armed(P) after dunk(P,T).
          caused dunked(P) after dunk(P,T).
          caused unsafe if armed(P).
          executable dunk(P,T) if not dunked(P).
initially:total armed(P).
          caused false if armed(P), armed(P1), P <> P1.
goal:     not unsafe?
```

Again, like in the hanoi example we use a fluent `unsafe` to express, that the goal has not yet been reached. `unsafe` remains true, as long as a package is `armed`.

We are allowed to `dunk` a package only once, that is why the fluent `dunked` is necessary to express that a package already has been dunked. This fluent is not really inertial, as there is no possibility to undo dunking, but simply remains true, whenever a package has been dunked, which is expressed with the second rule in the always section.

As we allow concurrent actions in this first formulation (order of dunking packages does not matter), the problem can always be solved in one step by dunking all packages into the toilet.

Assuming we have only one toilet bowl, the background knowledge for this domain can be defined as follows, allowing us to give the number of packages p by using option -N=p on the command line:

```
toilet(t1).
package(P) :- #int(P), P > 0.
```

For instance, having two packages, call `DLV` with the command:

```
$ dl bt.plan bt.bk -FP -N=2 -planlength=1
```

This yields the following optimistic plans:

```
PLAN: dunk(1,t1,0), dunk(2,t1,0)
PLAN: dunk(2,t1,0)
PLAN: dunk(1,t1,0), dunk(2,t1,0)
PLAN: dunk(1,t1,0)
PLAN: dunk(1,t1,0)
PLAN:
PLAN: dunk(2,t1,0)
PLAN: dunk(1,t1,0), dunk(2,t1,0)
```

where only the plans where both packages are dunked are secure of course, and the empty plan results from guessing that there is no armed package at all.

BTC(p) - Bomb in toilet with clogging Now we will observe the same domain with clogging, still assuming deterministic effects, i.e. dunking a package always clogs the toilet and we have to flush it.

We have to add a new fluent `clogged` and a new action `flush`:

```
fluents:  clogged(T) requires toilet(T).
          ⋮
actions:  flush(T) requires toilet(T).
```

and the following rules to the always section:

```
caused -clogged(T) after flush(T).
caused clogged(T) after dunk(P,T).

executable flush(T).
% modify executability of action dunk:
executable dunk(P,T) if not dunked(P), not clogged(T).

noConcurrency.
```

The executability of `dunk` now depends on the toilet not to be clogged. Concurrent actions cannot be allowed any longer, as dunking clogs the toilet, we always have to flush it in between and we are only interested in secure plans. For two packages we get two secure plans:

```
$ dl btc.plan bt.bk -FPsec -N=2 -planlength=1

PLAN: dunk(2,t1,0), flush(t1,1), dunk(1,t1,2)
PLAN: dunk(2,t1,0), flush(t1,1), dunk(1,t1,2)
PLAN: dunk(2,t1,0), flush(t1,1), dunk(1,t1,2)
PLAN: dunk(1,t1,0), flush(t1,1), dunk(2,t1,2)
PLAN: dunk(1,t1,0), flush(t1,1), dunk(2,t1,2)
PLAN: dunk(1,t1,0), flush(t1,1), dunk(2,t1,2)
```

Each plans is found three times in whole, representing the possible initial states, so we gain six plans found in total.

BMTC(p,t) - Bomb in toilet with clogging and multiple toilets The next generalization to be observed is the case of multiple toilets (t is the number of toilets). To take advantage of multiple toilets we cannot simply use `noConcurrency.` like above but have to declare explicitly, when concurrent actions are allowed, by adding the following constraints to the always section instead:

```
% instead of noConcurrency:
nonexecutable flush(T) if dunk(P,T).
nonexecutable dunk(P,T) if dunk(P1,T), P <> P1.
nonexecutable dunk(P,T) if dunk(P,T1), T <> T1.
```

As in the rocket domain, we can see here, how to declare possible action conflicts using `nonexecutable` statements. In addition to that, we have to add facts `toilet(t2). toilet(t3). ...` to the background knowledge for multiple toilets.

For instance, for BMTC(4,2) DLV$^{\mathcal{K}}$ finds 24 secure plans (12 possibilities for dunking two packages in the first step, flushing both toilets in the second step, and two possibilities of dunking the remaining two packages in the third step). Each plan is found five times due to the five possible legal initial states, thus we have 120 resulting plans calling `dl`:[3]

```
$ dl bmtc.plan bt_2t.bk -N=4 -planlength=3 -FPsec

PLAN: dunk(3,t1,0), dunk(4,t2,0), flush(t1,1), flush(t2,1),
dunk(1,t1,2), dunk(2,t2,2)
PLAN: dunk(3,t1,0), dunk(4,t2,0), flush(t1,1), flush(t2,1),
dunk(2,t1,2), dunk(1,t2,2)
⋮
```

BTUC(p,t) - Bomb in toilet with uncertain clogging Finally we will observe the domain, where clogging is an uncertain outcome of dunking a package. To achieve this, the clog effect of action `dunk` has to be modified to be `total`:

```
total clogged(T) after dunk(P,T).
```

For one toilet and 2 packages we need at least three steps and find two secure plans:

```
PLAN: dunk(2,t1,0), flush(t1,1), dunk(1,t1,2)
PLAN: dunk(1,t1,0), flush(t1,1), dunk(2,t1,2)
```

where again, each plan is found multiple times due to the different possible initial and intermediate states (3 possible initial states, 2 possible dunks in the first step, 2 possible result of dunk, yields 12 results). The final version of BTUC(p,t) is shown in figure 6.7.

As mentioned before, there are possible problems in the current implementation for totality in the always section of a problem. Let us take a closer look now why the presented encoding of BTUC(p,t) works fine. The only nondeterministic effect is clogging in the rule:

```
total clogged(T) after dunk(P,T).
```

[3]Note that the number of extracted plans is not monotonic with the number of packages in this problem, as for instance BMTC(3,2) results in 576 (24^2) possible plans For two toilets, an odd number of packages, does not enforce the agent to dunk a maximum number of packages in every step. whereas the only possibility to reach the goal for BMTC(4,2) in three steps is to dunk 2 packages in the first and 2 packages in the third step, while flushing the toilets in between.

```
fluents:  armed(P) requires package(P).
          clogged(T) requires toilet(T).
          dunked(P) requires package(P).
          unsafe.
actions:  dunk(P,T) requires package(P), toilet(T).
          flush(T) requires toilet(T).
always:   inertial armed(P).
          inertial clogged(T).
          caused dunked(P) after dunked(P).
          caused -clogged(T) after flush(T).
          caused -armed(P) after dunk(P,T).
          total clogged(T) after dunk(P,T).
          caused unsafe if armed(P).
          executable flush(T).
          executable dunk(P,T) if not dunked(P), not clogged(T).
          % instead of noConcurrency:
          nonexecutable dunk(P,T) if flush(T).
          nonexecutable dunk(P,T) if dunk(P1,T), P <> P1.
          nonexecutable dunk(P,T) if dunk(P,T1), P <> T1.
initially:total armed(P).
          caused false if armed(P), armed(P1), P <> P1.
goal:     not unsafe?
```

Figure 6.7: `btuc.plan`: $\text{DLV}^{\mathcal{K}}$ representation of BTUC(p,t)

Problems could occur in the successor state, whenever a constraint is violated or a possible inconsistency arises because of the nondeterministic effect. The only possible inconsistency is due to the `-clogged(T)` effect of action `flush(T)`, which could conflict with the possible `clogged(T)` effect of action `dunk(P,T)`. However, the concurrent execution of `flush` and `dunk` is forbidden by a constraint. Hence a conflicting situation like shown in Section 4.2.2 cannot occur in this domain, and all secure plans will be found by $\texttt{DLV}^{\mathcal{K}}$.

Chapter 7

Performance of DLV$^{\mathcal{K}}$

In this chapter, DLV$^{\mathcal{K}}$ will be compared with *CCALC*, a system accepting $\mathcal{C}$ action language as input. The results presented here are not intended to present sheer benchmarks but more to give a momentary view on the state of the current implementation and its capabilities.

7.1 Comparing DLV$^{\mathcal{K}}$ with *CCALC*

7.1.1 *CCALC*

The *Causal Calculator* (*CCALC*) is a model checker for the languages of causal theories [40]. It translates the $\mathcal{C}$ action language into the language of causal theories which are in turn transformed into a SAT problem using literal completion as described in [41]. This approach is based on Satisfiability Planning [33].

The system can be downloaded at [39]. Planning problems are reduced to a SAT-problem which is then solved, using an efficient solver like *SATO* (see [59] or *relsat* [3]). In [19] and [20] experimental results for the blocks world domain can be found, which are compared to DLV$^{\mathcal{K}}$ subsequently. The blocks world problems P1-P4, which are examined here, are extensions of the Sussman Anomaly (Figure 6.1), taken from these experiments.

We have tested *CCALC* only with deterministic domains. Though there is a plan verification option in *CCALC*, it can not deal with nondeterminism of any kind in terms of conformant planning. Verification always fails in presence of incomplete knowledge, so it seems only to be capable of checking determinism of the domain. So, what *CCALC* calculates are "optimistic plans" following $\mathcal{K}$ terminology.

For $\mathcal{C}$ language input there exists another system based on *-SAT described in [29], [24], which seems to be faster and capable of conformant planning as well. Unfortunately, we could not obtain the system for testing so far.

A $\mathcal{C}$ encoding of $BTUC(p)$ is given in the appendix.

7.1.2 Test Environment

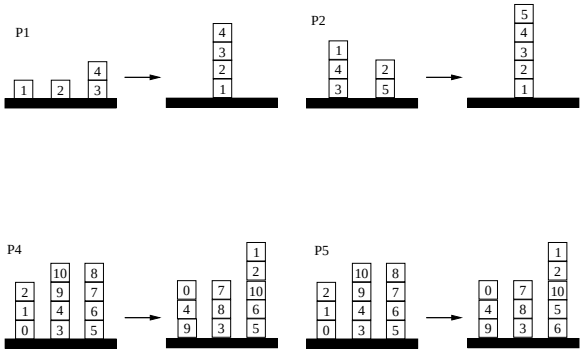
All the tests presented here were performed on a `Pentium II 350MHz/256MB RAM` machine running Red Hat Linux 6.2. *CCALC* was tested with the free Prolog systems `GNU Prolog 1.2.1`[1] and `SWI-Prolog 3.4.4`[2] using `SATO 3.2.1`[3].

Unfortunately, *relsat*, the alternative SAT solver used by *CCALC*, did not compile under Linux, nor did the pre-compiled binary for Solaris run at any machine available.

[1] `GNU Prolog` can be downloaded at `http://www.gnu.org`
[2] `SWI-Prolog` can be downloaded at `http://www.swi.psy.uva.nl/projects/SWI-Prolog/`
[3] `SATO` can be downloaded at `http://www.cs.uiowa.edu/~hzhang/sato.html`

7.2 Experimental Results for Deterministic Domains

The encodings used for DLV$^{\mathcal{K}}$ and *CCALC* for experimental evaluation can be found in the appendix. Problems P1-P4 are due to [19], and problem P5 is a slight modification of P4, which needs 2 steps more. The initial and goal configurations are shown in Figure 7.1



Problem	blocks	steps
P1	4	4
P2	5	6
P3	8	8
P4	11	9
P5	11	11

Figure 7.1: Some blocks world instances

Table 7.1 shows the performance of DLV$^{\mathcal{K}}$ and *CCALC* on these problems. The second column contains the minimum planlength to achieve the goal. The next columns `dl1` and `dl` denote the time to find the first plan and all plans resp. The number of plans found by DLV$^{\mathcal{K}}$ follows.

CCALC always displays only one model/plan. The timings for *CCALC* running under `GNU Prolog` and `SWI Prolog` respectively are shown in the last two columns. The results in parentheses (*grounding+completion+sato*) denote separated times for grounding, literal completion and calling SATO.

	$\|P\|$	`dl1`	`dl`	#PLANS	$CCALC_{swi,SATO}$	$CCALC_{gnu,SATO}$
P1	4	0.47s	0.51s	2	0.99 (0.54+0.42+0.03)	1.55 (0.73+0.80+0.02)
P2	6	1.55s	1.63s	3	2.28 (1.07+1.14+0.07)	2.43 (1.05+1.30+0.08)
P3	8	122s	333s	28	26.5 (9.97+15.38+1.15)	9.61 (3.48+4.99+1.14)
P4	9	110s	370s	2	259.5 (84.96+167.51+7.06)	30.15 (9.43+13.93+6.79)
P5	11	451s	3218s	2	??? (85.4+170.4+???)	??? (9.41+13.94+???)

Table 7.1: Experimental results for P1-P5 running DLV$^{\mathcal{K}}$ and *CCALC*

In general *CCALC* did perform better on the first problems, not solution for P5 could be found within 10 hours, which might also be a problem of the underlying SAT-Solver, as the grounding and literal completion were still performed rather quick, especially running *CCALC* with SWI Prolog.

Another remarkable result is, that DLV$^{\mathcal{K}}$ took less time for finding the first plan for P4 than for P3, which can probably be explained by the higher number

of models/plans for the latter.

Note that with a manual encoding of the blocks-world domain in the DLV core language, which takes advantage of the binarization split technique described in Section 4.2.3 the problem is solved much faster by DLV. This manual encoding is shown in Figure 7.2. This version outperforms the current implementation without binarization as well as *CCALC* by orders of magnitude, as we can see in Table 7.2.

	$\|P\|$	dl -n=1	dl	#PLANS
P1	4	0.28s	0.29s	2
P2	6	0.67s	0.72s	3
P3	8	3.33s	25.8s	28
P4	9	8.60s	12.5s	2
P5	11	14.7s	40.5s	2

Table 7.2: Experimental results for P1-P5 using split optimization.

7.3 Experimental Results for Nondeterministic Domains

Finally we will examine the performance of DLV$^{\mathcal{K}}$ with some nondeterministic domains, taking the bomb in the toilet examples from Section 6.7.5. The encodings used for these tests can can be found in the appendix.

$BT(p)$	$\|P\|$	dl1	dl	#PLANS
$BT(1)$	1	0.02s	0.03s	2
$BT(2)$	1	0.04s	0.07s	3
$BT(3)$	1	0.05s	0.16s	4
$BT(4)$	1	0.05s	0.45s	5
$BT(5)$	1	0.07s	1.25s	6
$BT(6)$	1	0.08s	3.47s	7
$BT(7)$	1	0.09s	9.59s	8
$BT(8)$	1	0.10s	25.5s	9
$BT(9)$	1	0.12s	65.3s	10
$BT(10)$	1	0.13s	165s	11

Experimental results for $BT(p)$

$BMTC(p,t)$	$\|P\|$	dl1	dl	#PLANS
$BMTC(1,2)$	1	0.06s	0.06s	8
$BMTC(2,2)$	1	0.09s	0.79s	6
$BMTC(3,2)$	3	1.20s	110s	576
$BMTC(4,2)$	3	28.6s	349s	120
$BMTC(5,2)$	5	57.6s	-	-
$BMTC(6,2)$	5	430s	-	720
$BMTC(1,3)$	1	0.06s	0.25s	24
$BMTC(2,3)$	1	0.08s	0.91s	36
$BMTC(3,3)$	1	0.10s	2.96s	24
$BMTC(4,3)$	3	0.80s	-	-
$BMTC(5,3)$	3	3.69s	-	-
$BMTC(6,3)$	3	18.3s	-	-

Experimental results for $BMTC(p,t)$

$BTC(p)$	$\|P\|$	dl1	dl	#PLANS
$BTC(1)$	1	0.06s	0.06s	2
$BTC(2)$	3	0.09s	0.79s	6
$BTC(3)$	5	1.20s	31.6s	24
$BTC(4)$	7	28.5s	1366s	120
$BTC(5)$	9	843s	-	720

Experimental results for $BTC(p)$

$BTUC(p)$	$\|P\|$	dl1	dl	#PLANS
$BTUC(1)$	1	0.04s	0.07s	4
$BTUC(2)$	3	0.13s	2.61s	24
$BTUC(3)$	5	4.41s	324s	192
$BTUC(4)$	7	396s	-	-

Experimental results for $BTUC(p,t)$

Table 7.3: Some results on the bomb in the toilet domain

The performance of DLV$^{\mathcal{K}}$ on these examples especially for finding **all** possible plans gets quite slow on increasing uncertainty, where the number of possible plans/models explodes. Thus, in the current state the system does not seem to

```
actiontime(T) :- T < #maxint, #int(T).
location(table).
location(B) :- block(B).
occupied(B,T) :- on(B1,B,T), block(B).
move_1(B,T) v -move_1(B,T) :- not occupied(B,T), block(B),
actiontime(T).
move_2(L,T) v -move_2(L,T) :- not occupied(L,T), B <> L,
location(L), move_1(B,T).
:- move_1(B,T), move_1(B1,T), B<>B1.
:- move_2(L,T), move_2(L1,T), L<>L1.
action(T) :- move_2(L,T).
:- not action(T), actiontime(T).

on(B,L,T1) :- on(B,L,T), not -on(B,L,T1), #succ(T,T1).
on(B,L,T1) :- move_1(B,T), move_2(L,T), #succ(T,T1).
-on(B,L,T1) :- move_1(B,T), not move_2(L,T), on(B,L,T),
#succ(T,T1).

% initially:
:- on(B,L,0), on(B,L1,0), L<>L1.
:- on(B1,B,0), on(B2,B,0), block(B), B1<>B2.
supported(B,0) :- on(B,table,0).  supported(B,0) :- on(B,B1,0),
supported(B1,0).
:- not supported(B,0), block(B).
```

Figure 7.2: Manual DLV encoding of blocksworld domain using split optimization.

be competitive with systems specially designed for conformant planning comparing the results in [11].

Nevertheless, the system keeps developing and improvements are expected. For instance, as for the blocksworld examples, a manual encoding exploiting binarization is shown below (Figure 7.3). The results cannot be compared directly with DLV$^{\mathcal{K}}$, as on the manual encoding no secure check can be performed. However, the times for optimistic planning can be compared, which includes all plans/models. Improvements can be noticed here as well:

The second column in the Table 7.4 denotes the time DLV$^{\mathcal{K}}$ takes to find all optimistic plans for the encoding in $\mathcal{K}$ using the normal translation for optimistic planning without checking security. The next column shows the time using the binarized encoding, which is also faster here, as for the deterministic domain. However speedup is not as tremendous as above.

Especially remarkable is the last column containing the number of **optimistic** plans computed by DLV: Obviously this number increases rapidly for added uncertainty. For secure planning each of this plans has to be checked, that means the ground program has to be rewritten and another instance of the model checker has to be called. This might partly explain, why the results for the bomb in toilet problem in the Tables 7.3 are rather slow.

$BTUC(p)$	$\|P\|$	dl -FPopt btuc.plan	dl btuc_bin.dl	#PLANS
$BTUC(1)$	1	0.05s	0.07s	6
$BTUC(2)$	3	0.39s	0.29s	188
$BTUC(3)$	5	23.6s	13.4s	7820
$BTUC(4)$	7	1781s	994s	407256

Table 7.4: Experimental results for BTUC using split optimization.

```
actiontime(X) :- T < #maxint, #int(X).
armed(P,X1) :- armed(P,X), not -armed(P,X1), #succ(X,X1).
clogged(T,X1) :- clogged(T,X), not -clogged(T,X1), #succ(X,X1).
dunked(P, X1) :- dunked(P,X),#succ(X,X1).
-clogged(T,X1) :- flush(T,X), #succ(X,X1).  -armed(P,X1)
:- dunk_1(P,X),#succ(X,X1).  dunked(P,X1) :- dunk_1(P,X),
#succ(X,X1).  clogged(T,X1) v -clogged(T,X1) :- dunk_2(T,X),
#succ(X,X1).
unsafe(X) :- armed(P,X).
flush(T,X) v -flush(T,X) :- toilet(T), actiontime(X).
dunk_1(P,X) v -dunk_1(P,X) :- package(P), not
dunked(P,X),actiontime(X). dunk_2(T,X) v -dunk_2(T,X) :-
toilet(T), not clogged(T,X),dunk_1(P,X).
% instead of noConcurrency:  :- dunk_2(T,X), flush(T,X). :-
dunk_1(P,X), dunk_1(P1,X), P<>P1.  :- dunk_2(T,X), dunk_2(T1,X), T
<> T1.
dunk(X) :- dunk_2(T,X). :- dunk_1(P,X), not dunk(X).
%initially:  armed(P,0) v -armed(P,0) :- package(P). :-
armed(P,0), armed(P1,0), P <> P1.
% goal:  not unsafe(#maxint)?
```

Figure 7.3: Manual DLV encoding of BTUC domain (btuc_bin.dl).

Chapter 8

Conclusions and Outlook

In this thesis it has been shown that language $\mathcal{K}$ is very expressive in terms of planning and reasoning about actions, allowing to encode even hard planning problems with incomplete initial knowledge, alternative preconditions of actions, and nondeterministic action effects.

With the current implementation of DLV$^{\mathcal{K}}$ we provide a system which allows a wide variety of domains to be encoded and solved.

However, also some limitations and possibilities for improvements and further research have developed throughout our work.

Theoretical results in Section 3.3 have shown that our language is indeed able to model harder problems than existing systems based on planning as satisfiability can solve. Nevertheless, so far only a small subset of proper planning domains can be checked for plan security, and another research aim might be to find a more general class of proper domains than stratified domains. Further results also support the assumption that looking for an efficient translation for general secure checking and the problem of secure plan existence for sequential planning might be rewarding.

Furthermore, it has been shown in Chapter 7 that there are possibilities in optimizing the translation by minimizing the grounded program under sequential planning, using database projection techniques. A further aim, especially when planning under uncertainty, should be to minimize the number of plans generated or the number of plans to be checked, respectively.

Bibliography

[1] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Los Altos, California, 1988.

[2] Chitta Baral, Vladik Kreinovich, and Raúl Trejo. Computational complexity of planning and approximate planning in presence of incompleteness. In *The International Joint Conferences on Artificial Intelligence (IJCAI) 1999*, pages 948–953, Stockholm, Sweden, August 1999.

[3] Roberto Bayardo and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-97)*, pages 203–208, 1997.

[4] R. Ben-Eliyahu and R. Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.

[5] Blai Bonet and Héctor Geffner. Planning with Incomplete Information as Heuristic Search in Belief Space. In Steve Chien, Subbarao Kambhampati, and Craig A. Knoblock, editors, *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS'00)*, pages 52–61, Breckenridge, Colorado, April 2000.

[6] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Adding Weak Constraints to Disjunctive Datalog. In *Proceedings of the 1997 Joint Conference on Declarative Programming APPIA-GULP-PRODE'97*, Grado, Italy, June 1997.

[7] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Strong and Weak Constraints in Disjunctive Datalog. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, number 1265 in Lecture Notes in AI (LNAI), pages 2–17, Dagstuhl, Germany, July 1997. Springer Verlag.

[8] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Enhancing Disjunctive Datalog by Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):845–860, 2000.

[9] T. Bylander. The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence*, 69:165–204, 1994.

[10] Alessandro Cimatti and Marco Roveri. Conformant Planning via Model Checking. In *Proceedings of the Fifth European Conference on Planning (ECP'99)*, September 1999.

[11] Alessandro Cimatti and Marco Roveri. Conformant Planning via Symbolic Model Checking. *Journal of Artificial Intelligence Research*, 13:305–338, 2000.

[12] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. In *Proceedings of the Twelfth Annual IEEE conference on Computational Complexity, June 24–27, 1997, Ulm, Germany, (CCC'97)*, pages 82–101. Computer Society Press, June 1997.

[13] T. Eiter and G. Gottlob. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Annals of Mathematics and Artificial Intelligence*, 15(3/4):289–323, 1995.

[14] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative Problem-Solving Using the DLV System. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

[15] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Planning under incomplete knowledge. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic - CL 2000, First International Conference, Proceedings*, number 1861 in Lecture Notes in AI (LNAI), pages 807–821, London, UK, July 2000. Springer Verlag.

[16] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Using the `dlv` system for planning and diagnostic reasoning. In François Bry, Ulrich Geske, and Dietmar Seipel, editors, *Proceedings of the 14th Workshop on Logic Programming (WLP'99)*, pages 125–134. GMD – Forschungszentrum Informationstechnik GmbH, Berlin, January 2000. ISSN 1435-2702.

[17] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Planning under incomplete knowledge. Extended version of [15]. To appear, 2002.

[18] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR System `dlv`: Progress Report, Comparisons and Benchmarks. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 406–417. Morgan Kaufmann Publishers, 1998.

[19] Esra Erdem. Applications of Logic Programming to Planning: Computational Experiments. Unpublished draft. `http://www.cs.utexas.edu/users/esra/papers.html`, 1999.

[20] Esra Erdem. Website for applications of logic programming to planning: Computational experiments, since 1998. `<URL:http://www.cs.utexas.edu/users/esra/experiments/experiments.html>`.

[21] Wolfgang Faber. Disjunctive datalog with strong and weak constraints: Representational and computational issues. Master's thesis, Institut für Informationssysteme, Technische Universität Wien, 1998.

[22] Wolfgang Faber, Nicola Leone, Cristinel Mateis, and Gerald Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. In INAP Organizing Committee, editor, *Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99)*, pages 135–139. Prolog Association of Japan, September 1999.

[23] Wolfgang Faber and Gerald Pfeifer. `dlv` homepage, since 1996. `http://www.dbai.tuwien.ac.at/proj/dlv/`.

[24] Paolo Ferraris and Enrico Giunchiglia. Planning as Satisfiability in Nondeterministic Domains. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI'00)*, Austin, Texas, 2000.

[25] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[26] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[27] Michael Gelfond and Vladimir Lifschitz. Representing Action and Change by Logic Programs. *Journal of Logic Programming*, 17:301–321, 1993.

[28] Michael Gelfond and Vladimir Lifschitz. Action languages. *Electronic Transactions on Artificial Intelligence*, 3(16), 1998.

[29] Enrico Giunchiglia. Planning as Satisfiability with Expressive Action Languages: Concurrency, Constraints and Nondeterminism. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR2000)*, pages 657–666, Breckenridge, Colorado, USA, April 2000. Morgan Kaufmann Publishers, Inc.

[30] Enrico Giunchiglia and Vladimir Lifschitz. An Action Language Based on Causal Explanation: Preliminary Report. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI '98)*, pages 623–630, 1998.

[31] Enrico Giunchiglia and Vladimir Lifschitz. Action languages, temporal action logics and the situation calculus. In *Working Notes of the IJCAI'99 Workshop on Nonmonotonic Reasoning, Action, and Change*, 1999.

[32] Steve Hanks and Drew McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[33] Henry Kautz and Bart Selman. Planning as Satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359–363, 1992.

[34] Henry Kautz and Bart Selman. Unifying sat-based and graph-based planning. In *The International Joint Conferences on Artificial Intelligence (IJCAI) 1999*, pages 318–325, Stockholm, Sweden, August 1999.

[35] V. Lifschitz and H. Turner. Representing transition systems by logic programs. In Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, number 1730 in Lecture Notes in AI (LNAI), pages 92–106, El Paso, Texas, USA, December 1999. Springer Verlag.

[36] Vladimir Lifschitz. Action Languages, Answer Sets and Planning. In K. Apt, V. W. Marek, M. Truszczyński, and D. S. Warren, editors, *The Logic Programming Paradigm – A 25-Year Perspective*, pages 357–373. Springer Verlag, 1999.

[37] Vladimir Lifschitz. Answer set planning. In Danny De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 23–37, Las Cruces, New Mexico, USA, November 1999. The MIT Press.

[38] J.W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, 1984.

[39] Norman McCain. The clausal calculator homepage, 1999. `<URL:http://www.cs.utexas.edu/users/tag/cc/>`.

[40] Norman McCain and Hudson Turner. Causal theories of actions and change. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-97)*, pages 460–465, 1997.

[41] Norman McCain and Hudson Turner. Satisfiability planning with causal theories. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 212–223. Morgan Kaufmann Publishers, 1998.

[42] John McCarthy. *Formalization of common sense, papers by John McCarthy edited by V. Lifschitz.* Ablex, 1990.

[43] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969. reprinted in [42].

[44] Drew McDermott. A critique of pure reason. *Computational Intelligence*, 3:151–237, 1987. cited in [11].

[45] Ilkka Niemelä. Logic programming with stable model semantics as constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999.

[46] Ilkka Niemelä and Patrik Simons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in AI (LNAI)*, pages 420–429, Dagstuhl, Germany, July 1997. Springer Verlag.

[47] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[48] Christos H. Papadimitriou and Martha Sideri. Default theories that always have extensions. *Artificial Intelligence*, 69:347–357, 1994.

[49] Gerald Pfeifer. Disjunctive Datalog — An Implementation by Resolution. Master's thesis, Institut für Informationssysteme, Technische Universität Wien, Wien, Österreich, 1996. Supported by Thomas Eiter.

[50] Raymond Reiter. A Logic for Default Reasoning. *Artificial Intelligence*, 13(1–2):81–132, 1980.

[51] Stuart J. Russel and Peter Norvig. *Artificial Intelligence, A Modern Approach.* Prentice-Hall, Inc., 1995.

[52] Patrick Simons. Smodels homepage, 1999. `<URL:http://www.tcs.hut.fi/Software/smodels/>`.

[53] David E. Smith and Daniel S. Weld. Conformant Graphplan. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98*, pages 889–896, July 1998.

[54] V.S. Subrahmanian and Carlo Zaniolo. Relating Stable Models and AI Planning Domains. In Leon Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 233–247, Tokyo, Japan, June 1995. MIT Press.

[55] Gerald J. Sussman. The Virtuous Nature of Bugs. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, chapter 3, pages 111–117. Morgan Kaufmann Publishers, Inc., 1990. originally written 1974.

[56] J. D. Ullman. *Principles of Database and Knowledge Base Systems*, volume 1. Computer Science Press, 1989.

[57] Manuela Veloso. Nonlinear problem solving using intelligent causal-commitment. Technical Report CMU-CS-89-210, Carnegie Mellon University, 1989.

[58] C. Zaniolo, S. Ceri, C.Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems.* Morgan Kaufmann Publishers, Inc., 1997.

[59] Hantao Zhang. SATO: An Efficient Propositional Prover. In *Proceedings of the International Conference on Automated Deduction (CADE'1997)*, pages 272–275, 1997.

Appendix A

Input Files for Experiments

A.1 Input Files for *CCALC*

```
:- include 'C.t'.
:- sorts location >> block.


:- variables B, B1, B2 ::  block;
             L, L1 ::  location;
             A, A1 ::  action.
:- constants table ::  location;
             on(block, location) ::  inertialTrueFluent;
             above(block, location) ::  defaultFalseFluent;
             move(block, location) ::  action.


:- show none.

move(B,L) causes on(B,L).
nonexecutable move(B,L) if on(B1,B).
never (on(B1,B) && on(B2,B) && -(B1=B2)).
caused -on(B,L1) if (on(B,L) && -(L=L1)).
caused above(B,L) if on(B,L).
caused above(B,L) if on(B,B1) && above(B1,L).
caused false if above(B,B).
caused false if -above(B,table).
```

Figure A.1: bw_domain.t (taken from [20])

```
:- include 'bw_domain.t'.
noconcurrency.
:- constants b0, b1, b2, b3 ::  block.
:- plan label::0;
facts::
0:  on(b0, table),
0:  on(b1, table),
0:  on(b2, table),
0:  on(b3,b2);
goal::
4:  (on(b0, table) && on(b1, b0) && on(b2, b1) && on(b3, b2)).
```

Figure A.2: bw_P1.t (taken from [20])

```
:- include 'C.t'.
:- sorts package.
:- sorts toilet.


:- variables P, P1 ::  package;
             T ::  toilet;
             A, A1 ::  action.
:- constants armed(package) ::  inertialFluent;
             dunked(package) ::  inertialFluent;
             clogged(toilet) ::  inertialTrueFluent;
             dunk(package,toilet) ::  action.


:- show none.
noconcurrency.
nonexecutable dunk(P,T) if dunked(P).
nonexecutable dunk(P,T) if clogged(T).
dunk(P,T) may cause clogged(T).
dunk(P,T) may cause -clogged(T).
dunk(P,T) causes -armed(P).
dunk(P,T) causes dunked(P).
```

Figure A.3: bw_BTUC.t (Encoding of bomb in toilet with uncertain clogging)

A.2 Input Files for DLV$^{\mathcal{K}}$

```
actions:
move(B,L) requires block(B), location(L).
fluents:
on(B,L) requires block(B), location(L).
occupied(B) requires location(B).
supported(B) requires block(B).
always:
executable move(B,L) if not occupied(B), not occupied(L), B <> L.
inertial on(B,L).
caused occupied(B) if on(B1,B), block(B).
caused on(B,L) after move(B,L).
caused -on(B,L1) after move(B,L), on(B,L1), L <> L1.
initially:
caused false if on(B,L), on(B,L1), L<>L1.
caused false if on(B1,B), on(B2,B), block(B), B1<>B2.
caused supported(B) if on(B,table).
caused supported(B) if on(B,B1), supported(B1).
caused false if not supported(B).
noConcurrency.
```

Figure A.4: blocks.plan

```
block(B) :- #int(B).
location(table).
location(B) :- block(B).
```

Figure A.5: P.bk

```
initially:
on(0, table).  on(1, table).  on(2, table).  on(3, 2).
goal:
on(0, table), on(1, 0), on(2, 1), on(3, 2)?
```

Figure A.6: P1.plan

```
toilet(t1).
% uncomment following lines for BMTC(p,2), BMTC(p,3):
% toilet(t2).
% toilet(t3).
package(P) :- #int(P), P > 0.
```

Figure A.7: bt.bk

```
fluents:
armed(P) requires package(P). dunked(P) requires package(P). unsafe.
actions:
dunk(P,T) requires package(P), toilet(T).
always:
inertial armed(P).
caused dunked(P) after dunked(P).
caused -armed(P) after dunk(P,T).
caused dunked(P) after dunk(P,T).
caused unsafe if armed(P).
executable dunk(P,T) if not dunked(P).
initially:
total armed(P).
caused false if armed(P), armed(P1), P <> P1.
goal:
not unsafe?  (1)
```

Figure A.8: bt.plan

```
fluents:
clogged(T) requires toilet(T).
armed(P) requires package(P).
dunked(P) requires package(P).
unsafe.
actions:
dunk(P,T) requires package(P), toilet(T).
flush(T) requires toilet(T).
always:
inertial armed(P).
inertial clogged(T).
caused dunked(P) after dunked(P).
caused -clogged(T) after flush(T).
caused -armed(P) after dunk(P,T).
caused dunked(P) after dunk(P,T).
% uncomment the following line for BTC, BMTC:
% caused clogged(T) after dunk(P,T).
% uncomment the following line for BTUC:
total clogged(T) after dunk(P,T).
caused unsafe if armed(P).
executable flush(T).
executable dunk(P,T) if not clogged(T), not dunked(P).
nonexecutable dunk(P,T) if flush(T).
nonexecutable dunk(P,T) if dunk(P1,T), P <> P1.
nonexecutable dunk(P,T) if dunk(P,T1), T <> T1.
initially:
total armed(P).
caused false if armed(P), armed(P1), P <> P1.
goal:  not unsafe?
```

Figure A.9: btuc.plan: Input file for BTC,BMTC and BTUC problems